An embedded augmented reality system

Michael Groufsky

A thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Engineering
in
Electrical and Computer Engineering
at the
University of Canterbury,
Christchurch, New Zealand.

10 February 2011

ABSTRACT

This report describes an embedded system designed to support the development of embedded augmented reality applications. It includes an integrated camera and built-in graphics acceleration hardware. An example augmented reality application serves as a demonstration of how these features are accessed, as well as providing an indication of the performance of the device.

The embedded augmented reality development platform consists of the Gumstix Overo computer-on-module paired with the custom-built Overocam camera board. This device offers an ARM Cortex-A8 CPU running at 600 MHZ and 256 MB of RAM, along with the ability to capture VGA video at 30 frames per second. The device runs an operating system based on version 2.6.33 of the Linux kernel.

The main feature of the device is the OMAP3530 multimedia applications processor from Texas Instruments. In addition to the ARM CPU, it provides an on-board 2D/3D graphics accelerator and a digital signal processor. It also includes a built-in camera peripheral interface, reducing the complexity of the camera board design.

A working example of an augmented reality application is included as a demonstration of the device's capabilities. The application was designed to represent a basic augmented reality task: tracking a single marker and rendering a simple virtual object. It runs at around 8 frames per second when a marker is visible and 13 frames per second otherwise.

The result of the project is a self-contained computing platform for vision-based augmented reality. It may either be used as-is or customised with additional hardware peripherals, depending on the requirements of the developer.

CONTENTS

ACKNOWLEDGEMENTS

With thanks to
Dr Michael Hayes, ECE Dept.
(senior supervisor)
and
Dr Mark Billinghurst, HIT Lab NZ
(co-supervisor)

Chapter 1

INTRODUCTION

Embedded systems—self-contained, special-purpose computers—offer an alternative to standard desktop or laptop hardware when such systems would prove to be too cumbersome, power-hungry, or unsuitable for another reason. This thesis describes the development of an embedded system designed to serve as an extendable platform for augmented reality applications. It consists of a stand-alone embedded computer along with all the necessary drivers and libraries required to support vision-based augmented reality. The development platform enables the creation of lightweight portable augmented reality viewing devices.

Augmented reality refers to the process of modifying, or 'augmenting,' a user's view of their environment by inserting virtual objects into the scene [Azuma 1997]. This may be achieved by supplying the user with a head-mounted or handheld display that shows a digitally altered view. Ideally, the virtual objects should merge seamlessly with their real counterparts.

An important consideration in any augmented reality application is the method employed to calculate the user's viewpoint in relation to the real world [State et al 1996]. This information is used to ensure virtual objects are rendered with the correct position and orientation. A wide variety of tracking techniques exist but the one employed in the project uses a camera and computer vision algorithms to recognise custom markers.

In recent years, high-end smartphones have emerged as a viable platform for augmented reality applications [Zhou et al 2008]. By combining a camera and high-resolution display with a high-performance embedded processor these devices offer everything necessary for vision-based augmented reality in a compact and lightweight form-factor. The major advantage that platforms such as these offer is that they free the user of the restriction of being tethered to a regular computer [Schmalstieg and Wagner 2007].

One drawback of consumer electronics devices is that the hardware configuration is fixed by the manufacturer. Alternative configurations, such as a head-mounted display or multi-camera setup, are not available to the developer. In this situation, the developer must resort to a custom embedded system. The device described in this

thesis is intended to serve as a basis for such systems.

This thesis covers the following topics:

- A history of augmented reality, including an overview of the different techniques and technologies employed (Chapter 2).
- An evaluation of three different vision-based tracking techniques (Chapter 3).
- A discussion of the design decisions involved in developing an embedded system for vision-based augmented reality (Chapter 4).
- A description of a prototype embedded augmented reality device (Chapter 5).
- An example augmented reality application created to run on the prototype device (Chapter 6).

Chapter 2

BACKGROUND

2.1 INTRODUCTION

The field of augmented reality, or AR, has strong connections to virtual reality and it is helpful to compare these two technologies. Both involve the generation of three-dimensional virtual objects that are then inserted into the user's view of the world. However, where virtual reality systems aim for total immersion, completely replacing the user's surroundings with a computer-generated environment, augmented reality systems instead present a composite scene to the user, combining virtual objects with elements from the real world. The result is that the user experiences an *augmented* view of their environment [Azuma et al 2001].

Traditionally, AR research has focused on applications in areas such as medicine and manufacturing and repair [Azuma 1997]. In both of these cases, the goal has been to make professional workers more effective by overlaying information relevant to their job into their real work environment. In recent years, casual applications such as gaming and marketing are receiving increasing attention, in part due to the rise in popularity of powerful mobile phones able to support such applications [Ashley 2008].

A critical challenge in any AR system is to accurately establish the location of certain features in the local environment relative to the user of the system, so that the virtual objects may be drawn in their correct position and with their correct orientation. In vision-based AR, this has traditionally been achieved with the aid of markers introduced into the scene and computer vision tracking. Drawbacks of this approach (such as their reliance on well-lit environments) have lead to increasing interest in markerless systems, which work by tracking naturally-occurring features [Zhou et al 2008].

With few exceptions, until 2003 AR systems were built from standard workstations or, later, PC hardware. The normal approach to system assembly was to construct a head-mounted rig with a display and tracking system and having cables running to a stationary computer nearby. Interest in outdoor applications spurred the development of portable setups, with the computer carried on the user's back [Feiner et al 1997]. Around 2003, PDA and smartphone technology reached the point where these devices

became suitable AR platforms [Wagner and Schmalstieg 2003], enabling the emergence of handheld AR.

The tracking technologies and processing platforms covered in this chapter serve as a starting point for the design of a new augmented reality device. The project described in this thesis follows on from the work done on handheld AR, exploring the use of custom hardware in place of consumer devices (i.e., smartphones and PDAs). In particular, its design is informed by existing projects that make use of embedded systems such as those inside modern mobile phones.

Section 2.2 reviews the broad history of augmented reality research, drawing attention to specific projects and publications that have been influential in shaping this field. Following this, in Section 2.3, is an overview of the various solutions that have been employed to solve the tracking problem. In the Section 2.4, the evolution of augmented reality systems is considered from a hardware perspective, from early workstation-based setups to recent mobile phone applications.

2.2 HISTORY OF AUGMENTED REALITY

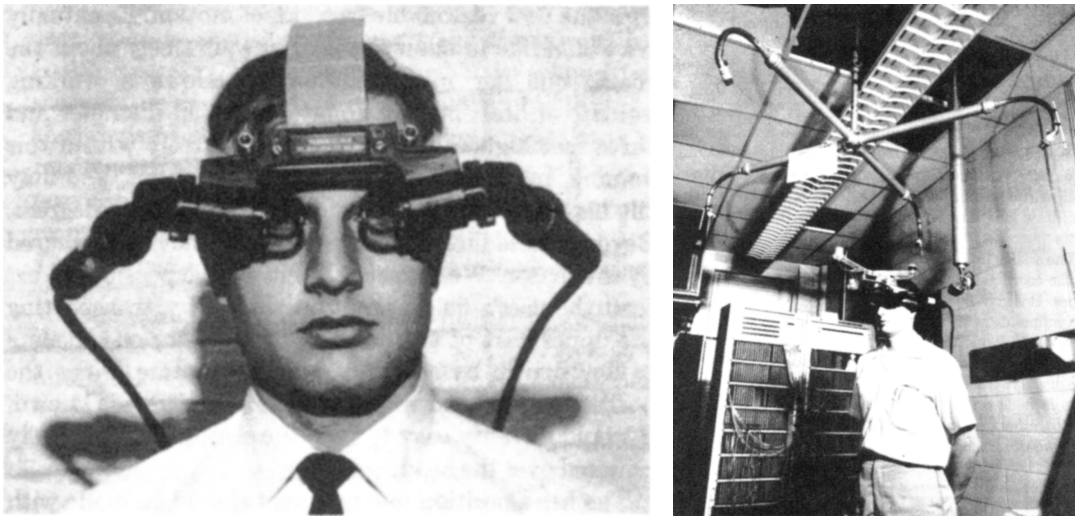
The first true augmented reality system was Ivan Sutherland's "head-mounted three-dimensional display" [Sutherland 1968], shown in use in Figure 2.1. With the aid of miniature CRT displays and either a mechanical or ultrasound-based head tracker, the user could examine or interact with 3D wireframe objects that were superimposed on their view. These objects could "be made either to hang disembodied in space or to coincide with maps, desk tops, walls, or the keys of a typewriter."

Though this work was carried out decades before the term "augmented reality" was introduced[1], it brought up a number of issues which would remain areas of intensive research in the years to come. In particular, the following questions had to be addressed in this—and indeed, any—AR system [Azuma 1997]:

1. How are the real and virtual elements to be combined?
2. How can the virtual objects be registered in 3D relative to the user and their surroundings?
3. How do we achieve "real-time" interactive performance?

The first issue concerns the key distinction between virtual and augmented reality: in an AR system, the view that the user is presented with is composed of both virtual and real elements. The technique that Sutherland used employed half-silvered mirrors in front of the user's eyes to superimpose the wireframe models on the user's view of

[1]The term "augmented reality" was coined by Tom Caudell and David Mizell in a 1992 paper [Caudell and Mizell 1992].



(a) Ivan Sutherland wearing his HMD (b) The HMD in use

Figure 2.1: Ivan Sutherland's head-mounted display from 1968. Image (a) shows the head-mounted display optics, with miniature CRTs. In image (b) the ultrasonic head position sensor is tracking the user's head. Behind the user is the mechanical arm that provides an alternative tracking method. (From Sutherland [1968].)

the real world. Other systems (e.g., [Bajura and Neumann 1995]) present the user with a live camera feed that has been altered to introduce virtual objects. These two approaches are discussed in more detail in Section 2.3.

The real-virtual combination problem is relatively straightforward; a more serious challenge is registration of the virtual objects, that they may be correctly aligned with the real world. In general, this involves tracking the position and orientation of the user's viewpoint. There are multiple techniques for obtaining this information. For example, Sutherland's head-mounted display employed both mechanical and ultrasound sensors for direct measurement of the user's head position. Many recent systems, however, make use of optical tracking, where computer vision techniques are applied to pick out features in the user's environment.

The final issue, achieving real-time interactive performance, comes up in many areas of computing research. There are two stages of the AR pipeline that require significant processing power: extracting useful information from the raw tracking data (particularly in the case of optical tracking), and rendering the scene that is presented to the user (see performance evaluations in Chapter 3). As in other areas, the strategy tends to be to seek out more efficient algorithms while also taking advantage of advances in hardware performance.

A variety of application domains have been explored that could potentially benefit from augmented reality technologies. For example, in the medical domain applications such as the one described by State et al. [State et al 1996] have been proposed to assist

medical professionals in their work. This particular system renders live ultrasound data in 3D, allowing a doctor to 'see inside' their patient. The objectives are similar in manufacturing and repair, as in the KARMA project [Feiner et al 1993], which included an application to guide a user through various maintenance tasks on a laser printer by highlighting the appropriate components in each step.

2.3 TRACKING AND DISPLAY TECHNIQUES

Over the years, many different techniques have been proposed to address the registration issue; that is, how to determine the position and orientation of the user relative to their environment, so that virtual objects may be rendered in the correct location relative to the real scene. These techniques range from direct-sensing methods, whereby the relevant data is measured directly, through to more sophisticated computer vision applications that extract this information from live image stream.

A variety of systems exist for presenting the augmented scene to the user. One option is to use a head-mounted display such as the one used by Ivan Sutherland. These come in both *optical see-through* and *video see-through* varieties. Other techniques that have been explored are projection-based systems, whereby a digital projector "paints" virtual objects onto the real environment (e.g., Raskar et al [1999]), and standalone displays (described later in this section).

An optical see-through head-mounted display is normally transparent to the user, or nearly so, with computer-generated imagery (i.e., virtual objects) inserted into the user's field of view. This is typically accomplished using a half-silvered mirror mounted in front of the user's eyes. Video see-through head-mounted displays provide the user with a composite image formed partly from computer-generated objects and partly from video data provided by a head-mounted camera.

Optical see-through systems have the advantage that the user's view of their environment is the real view, unmediated by digital capture and display. It does, however, place high demands on registration accuracy and latency, since even small misalignments or delays are immediately apparent [Azuma 1997]. Video see-through displays have the advantage that the live stream can be synchronised with the tracking process. Information from the video stream can also be used in a feedback process to dynamically correct registration error [Bajura and Neumann 1995].

A head-mounted display is just one possible configuration for an augmented reality system. A kiosk-style setup can be constructed from a fixed, free-standing display coupled with a suitable tracking technology, such as the one used in the MR Sea Creatures exhibit [Hughes et al 2004]. Another possibility is to make use of widely-available handheld devices incorporating a screen and camera; these platforms are discussed in Section 2.4. Figure 2.2 shows the benefits of such configurations: in the

(a) The MR Sea Creatures exhibit

(b) AR game using handheld devices

Figure 2.2: Two alternative AR configurations; (a) a kiosk-style setup with fixed screen (from http://e2i.ist.ucf.edu/) and (b) handheld multimedia devices (from Schmalstieg and Wagner [2007]).

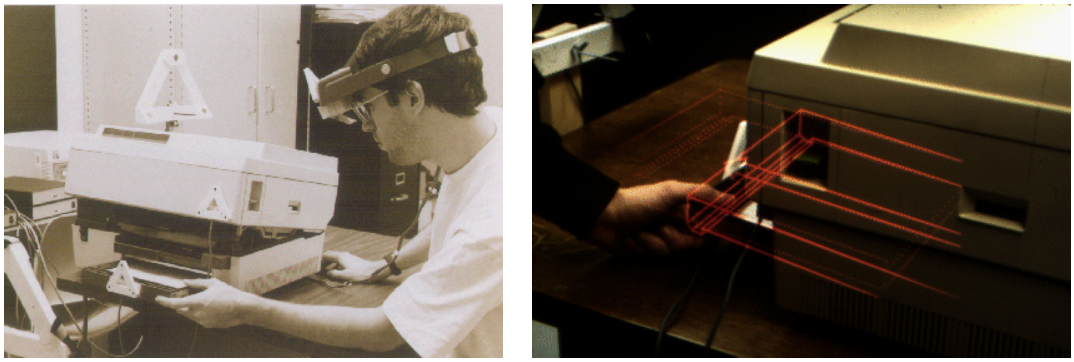
kiosk-type, a large display can be employed allowing more than one participant to take part, while small handheld devices can be used singly or shared with others as desired.

In addition to display technology there is a need for a tracking system that can supply the user's viewpoint. A common solution to the tracking problem in early systems was to deploy sensors to measure the location of key points (such as the user's head or an object held in his or her hands) directly, as in the case of Sutherland's head-mounted display mentioned previously. A variety of sensing technologies were employed, including magnetic, acoustic, inertial, optical, and mechanical [Zhou et al 2008]. Each of these has their own advantages and disadvantages that must be considered when deciding which technique to employ for a given application [Rolland et al 2001].

During the nineties, researchers in augmented reality could take advantage of developments in display and tracking technology spurred by the commercial interest in virtual reality [Zhou et al 2008]. In Figure 2.3, a user carries out maintenance on a printer with the aid of the KARMA system [Feiner et al 1993]. This setup is typical of the period, consisting of a head-mounted optical see-through display in conjunction with an off-the-shelf tracking device (in this case, an ultrasound-based 3D positioning system from Logitech). Multiple tracking technologies could be combined in order to compensate for weaknesses in one or the other [State et al 1996], although latency issues remained a challenge [Jacobs et al 1997].

With the increasing use of video (as opposed to optical) see-through displays came a variety of techniques that extracted useful information from the camera image itself. At first, this was applied in a weak sense, to correct tracking errors in a sensor-based system [Bajura and Neumann 1995]. Later, however, techniques were developed to track special markers or other objects, doing away with the need for additional sensors [Lepetit and Fua 2005].

One of the first examples of such systems was the one described in Rekimoto



(a) The KARMA head-mounted display (b) Overlaid graphics

Figure 2.3: The KARMA (Knowledge-based Augmented Reality for Maintenance Assistance) prototype system. Image (a) shows a user wearing the head-mounted display. Alongside the user, a laser printer is being tracked in 3D via an ultrasound-based system from Logitech; these are the triangular objects attached to it. The resulting augmented scene is shown in (b). (From http://graphics.cs.columbia.edu/projects/karma/karma.html.)

[1998]. In this system, square barcode-like "matrix" markers are affixed to physical objects. When these markers are visible to the camera, they serve both as a method of identifying the object and of registering it in three dimensions. This approach would later form the basis of the popular ARToolKit library [Kato and Billinghurst 1999] (ARToolKit is described in greater detail in Section 3.2).

Markers, also referred to as fiducials, provide a good solution to the tracking problem in a variety of situations, but are most well-suited to the 'augmented workspace' scenario [Sauer et al 2000]. This case typically involves a controlled indoor space, where reasonable lighting is guaranteed and markers can be set up as necessary. Under such conditions, markers are able to provide accurate tracking information in minimal processing time [Zhang et al 2002]. Specialised marker designs may go some way to relieving constraints such as range limitations [Cho et al 1998], however the use of markers in outdoor augmented reality remains impractical [Azuma et al 1999].

Markers supply a set of artificial features by which the system orients itself. An alternative approach is to take advantage of naturally-occurring features to serve the same purpose. Natural features present in the scene can be employed to stabilise and extend the tracking ranges of augmented reality systems [Neumann and You 1999]. They may be combined with markers [Subbarao et al 2005] or replace them entirely [Comport et al 2003].

One application of natural features is in the detection and registration of specific objects of interest expected to be present in the environment. These systems require a model of the object (say, a CAD model), from which key features are extracted. These features are correlated with those present in the scene in order to establish a mapping

Figure 2.4: The AR handheld display as used at the HIT Lab NZ. Visible in this picture are the display optics and the controls; the camera is situated on the far side of the device.

from one to the other. Model-based tracking is covered in Section 3.3.1.

Natural features may also be employed to facilitate extendible tracking techniques. This refers to the process of extending the tracking range beyond the initial frame of reference (typically a simple model [Bleser et al 2006] or configuration of markers [Subbarao et al 2005]) by incorporating detected features into the tracking procedure. Extendible tracking is described in Section 3.4.1.

2.4 PROCESSING PLATFORMS

A notable handheld display device, and an important inspiration for this project, was the one assembled for the MagicBook project [Billinghurst et al 2001]. It incorporated miniature LCD displays and a camera mounted at the top of a handle, by which the user grasped the device. On the handle were a set of controls, for user input. No processing was done on-board, so each component (the display, camera, and controls) connected to a nearby computer by a cable extending from the base of the handle.

Figure 2.4 shows an updated version of the device, currently in use at the HIT Lab NZ, and gives an idea of its scale and general configuration. In the author's experience there are two factors that limit the effectiveness of this device: its bulk, due to the use of common off-the-shelf components (e.g., a USB webcam for the camera), and its reliance on a fixed computer to which the device is tethered by its cables.

Early experiments with mobile (i.e., untethered) AR systems, such as the "Touring Machine" [Feiner et al 1997], consisted of a backpack computer attached to a head-

mounted display and tracking system. A typical task for such systems was to provide navigation aids in outdoor environments, using GPS for position tracking [Thomas et al 1998, Azuma et al 1999].

Being built from commodity PC hardware, there were limits on how small these systems could be made. A first look at a more lightweight alternative came in 1999, with the proposal of a custom embedded system based on multiple "StrongARM" processors [Pouwelse et al 1999]. The designers of this system calculated that it would weigh 2 kg (including 1 kg of batteries), only a tenth of the weight of their prototype made from commodity components. However, few examples of true embedded AR systems exist (see Section 4.2.1 for an overview of these) and serious investigation of AR on embedded devices did not occur until the emergence of modern camera phones.

Handheld consumer electronics devices such as PDAs and smartphones have for a long time held appeal as potential AR platforms due to their convenience and compactness [Wagner and Schmalstieg 2003]. Early efforts were constrained by the limited processing power of these devices; the AR-PDA project [Gausemeier et al 2003] had the PDA offload heavy processing to a nearby computer over a wireless link. Eventually, high-end mobile phones reached the point where it was practical to process the image and render the result entirely on the device itself [Möhring et al 2004, Henrysson et al 2005].

Driving up the power and complexity of these devices is strong consumer demand for more-capable mobile phones. The processors used tend to be of the "system-on-chip" type, integrating a variety of common peripherals such as an LCD controller and an image sensor interface on the same die as the processor core (most commonly an ARM).

The ARToolKitPlus library [Wagner and Schmalstieg 2007] was developed to provide a marker-based AR framework similar to ARToolKit on a variety of handheld devices, a selection of which are shown in Figure 2.5. More recently, natural-feature-based techniques have been made to run on high-end mobile phones [Wagner et al 2008a, Ta et al 2009].

2.5 SUMMARY

Modern Augmented Reality systems have a wide variety of tracking techniques and processing platforms to choose from. The hardware and software used in these systems is the result of many years of research. Where possible, techniques and technologies from related fields (such as virtual reality and computer vision) have been employed.

A relatively recent development is handheld AR using smartphones and similar self-contained devices. This offers the advantage of a viewing device that is lightweight and not restricted by cables that connect it to a fixed system. However, researchers

Figure 2.5: The ARToolKitPlus marker-based augmented reality system running on three different mobile devices. From left to right, these are: a smartphone, a PDA, and a tablet computer. (From Schmalstieg and Wagner [2007].)

working with these devices are limited to standard phone-type form factors and do not have the option of, for example, an eye-aligned display such as the one shown in Figure 2.4. The project described in this thesis makes use of a customisable hardware platform, leaving the choice of form-factor up to the system developer.

Chapter 3

EVALUATION OF DIFFERENT TRACKING APPROACHES

3.1 INTRODUCTION

The emergence of vision-based tracking techniques is one of the major advances in augented reality, replacing sensor-based approaches in many applications [Zhou et al 2008]. These techniques are often used in conjunction with video see-through displays, as the same camera can be used for both the tracking process and the user's view [Bajura and Neumann 1995]. While simplifying the hardware, the tradeoff is added complexity in the software due to the sophisticated tracking algorithms employed [Koller et al 1997].

A number of factors arise when considering the different visual tracking methods [Lepetit and Fua 2005]:

- Should it rely on artificial or natural features? Artificial features (i.e., markers) can be designed to be reliably detectable by the computer, but it is not always practical to place markers around the scene.

- A common approach is to attempt to track a known object (either a marker or an arbitrary model), determining its pose relative to the camera. Alternatively, the algorithm can assume an unknown but more-or-less static environment and attempt to calculate the location of the camera relative to this.

- The algorithm may assign a motion model to the tracked object and use it to estimate the object's position when detection fails. However, this adds another level of processing that is unnecessary in many cases.

The three tracking methods covered in this chapter comprise a representative sample of the different ways that these considerations have been addressed.

The first is the ARToolKit library, a typical marker-based solution to the tracking problem. Such systems are programmed to recognise a number of predefined markers.

Whenever one of these markers is detected in the image stream, its position and orientation relative to the camera is calculated and may then be used to render a virtual object.

In applications where markers are not suitable, tracking schemes may be designed that operate on naturally-occurring features instead. The other two methods covered in this chapter are reference implementations of this type of tracker. BazAR is an example of the template-based approach, where the system is trained ahead of time to detect a specific object. This object then plays the same role as a marker, providing 3D coordinates for the rendering of virtual objects.

The final method, PTAM, also employs a type of object model. However, instead of targeting a specific item, the model is of a generalised flat surface and is used only to get the initial frame of reference. The main task of the PTAM system is to estimate the camera motion relative to the environment, which is assumed to be more-or-less static. By keeping track of natural features and applying a motion model to the camera, PTAM can continue to provide camera coordinates even when the initial flat surface is no longer in view.

3.2 ARTOOLKIT

ARToolKit is a set of libraries for developing marker-based AR applications. It was initiated in 1999 as an in-house project at the Human Interface Technology Lab, University of Washington. The first versions were developed by Dr. Hirokazu Kato and Mark Billinghurst[1]; later (in 2004) it became an open-source project hosted on SourceForge.net.

The ARToolKit libraries provide routines for receiving a video stream from an attached camera, creating and updating a display window, and handling user input. This is in addition to the various functions that carry out marker detection, identification and registration. Graphics drawing operations are carried out by calls to a standard graphics library (OpenGL).

The application in Figure 3.1 is a video-based AR conferencing system [Kato and Billinghurst 1999]. This particular program was one of the first to be built with ARToolKit. Visible in the screenshot are two elements typical of such applications: the physical ARToolKit markers (recognisable by their black borders) and the virtual objects generated by the computer.

The ARToolKit distribution is made available for download from the developer website[2] under the GNU General Public License. This package contains the source code for the core ARToolKit libraries and a number of example programs and utilities,

[1] See http://www.hitl.washington.edu/artoolkit/documentation/history.htm for a full account of the history of ARToolKit.

[2] http://sourceforge.net/projects/artoolkit/

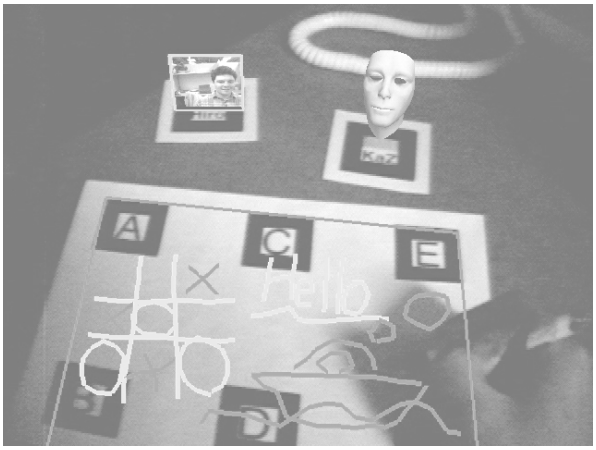


Figure 3.1: An early version of the ARToolKit is used to create a virtual shared whiteboard. The system has detected the markers in the foreground (identified by the characters A–E) and is using their positions to place the whiteboard overlay. In the background, two more markers have had virtual objects (a mask and a photograph) superimposed on them. (From Kato and Billinghurst [1999].)

along with build scripts targetting the Linux, Mac OS X and Windows platforms. Also included are a number of sample markers and marker templates.

An important feature of ARToolKit is that it runs on commodity computer hardware and provides a reasonable level of performance even on less powerful systems [Zhang et al 2002]. Unfortunately, this means that the user of such a system is tethered to a desktop computer, or else must carry a laptop around with them if they require mobility. This is obviously less than ideal, so a number of groups have been developing AR systems that run on handheld devices. Mobile phones are a particularly attractive target—most people have one, and most current models include an integrated camera and graphical display. Though they tend to be considerably less powerful than desktop computers, they are improving all the time, and recent high-end models are now capable of running serious AR applications [Schmalstieg and Wagner 2007].

3.2.1 Marker-based systems

The ARToolKit library is an implementation of a marker-based tracking system. This approach is popular in vision-based AR, since markers can be designed to provide reliably-detectable features by which the system can orient itself along with patterns or codes for marker identification. The marker design and tracking algorithm employed by the ARToolKit are typical of such systems.

The two important features of an ARToolKit marker are a square black border on a white background, used to calculate its pose, and a graphical symbol for identification. Many marker detection schemes make use of a square border (examples are shown in Figure 3.2) as it is easily isolated in an image and can be quickly processed to provide

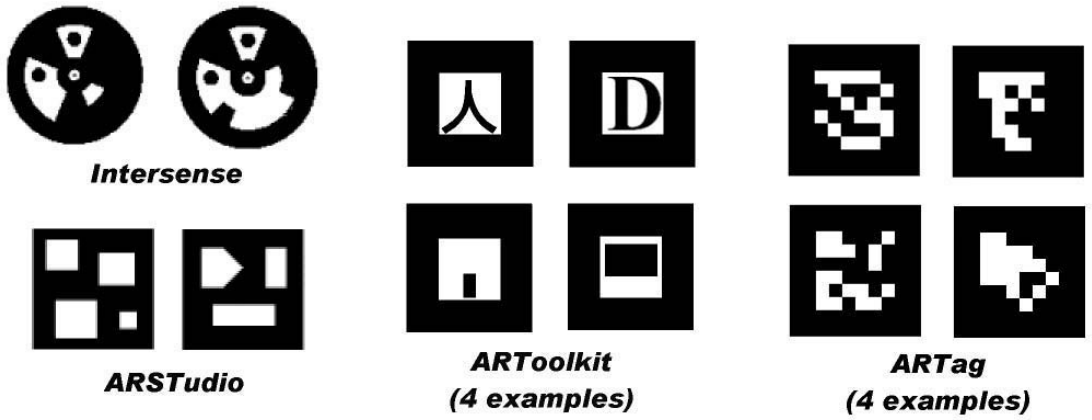


Figure 3.2: A selection of markers used in various marker-based AR systems. Identification data may be symbolic (ARSTudio, ARToolKit) or encoded (Intersense, ARTag). A strong circular or square border differentiates the marker from its surroundings. (Adapted from Fiala [2005a]).

accurate pose information [Zhang et al 2002]. This holds even on mobile devices with limited processing power [Wagner and Schmalstieg 2007]. Circular markers may be used instead in situations where only the position (and not the orientation) is required [Naimark and Foxlin 2002].

The processing pipeline for a single frame in an ARToolKit-powered system is presented in the flow chart shown in Figure 3.3. An important constraint is that this sequence of operations must be carried out in real-time, which in practical terms means that it must be fast enough that the user does not notice any significant lag. For example, to keep up with a video stream updating at a standard 30 frames per second, the tracker would have to calculate the camera pose in less than 33 ms.

Different techniques are employed to identify the various marker types. In an ARToolKit application, the central region of each marker contains a unique symbol that serves this purpose. Another approach is to assign each marker an identifying number and encode it in a 2D matrix pattern [Rekimoto 1998]. Symbolic identification allows the design to be customised to suggest the content associated with that marker; however, the highly simplified template-matching algorithm used by the ARToolKit causes a problem with false marker recognition in some situations [Zhang et al 2002]. The encoded ID technique allows a greater number of unique markers to be supported and recognised reliably [Fiala 2005a].

The algorithm used to detect and identify a marker imposes constraints on its visual design, resulting in markers that are functional but visually unappealing. This problem is often overlooked entirely because in most detection schemes the focus is on technical performance. The Studierstube Tracker[3] is one framework that does address

[3] http://studierstube.icg.tu-graz.ac.at/

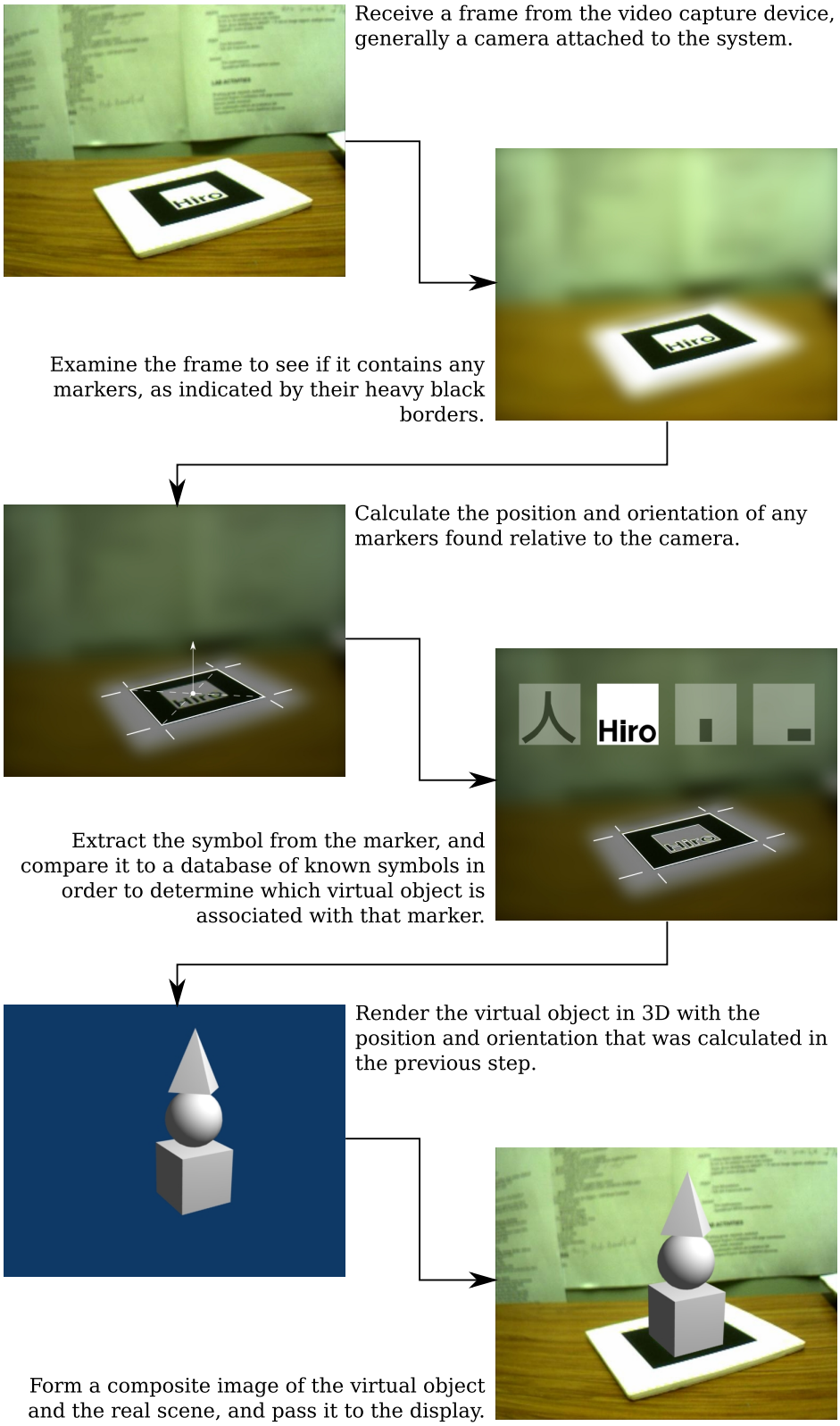


Figure 3.3: The processing pipeline of the ARToolKit for a single frame.

this issue, offering a variety of marker designs specifically formulated to be visually unobtrusive [Wagner et al 2008b].

The ARToolKit library relies on being able to detect the marker in every frame in which it appears; when the detection routine fails, it has no knowledge of the marker for that frame, thus is unable to augment the scene appropriately. This commonly occurs due to non-ideal viewing conditions such as the following [Ledermann et al 2002]:

- The marker is viewed from too far away or at too small an angle to be seen clearly.
- The marker is obscured by lighting effects such as shadow or glare.
- The marker is only partially visible, either because it is not completely within the field of view or it is occluded by intervening objects.

Of the three conditions described above, the occlusion issue is particularly problematic for systems that detect a marker by looking for the four edges that make up its border. If the border is interrupted at any point (say, by the thumb of the person holding the marker) then detection will fail [Fiala 2005b]. This limitation may be avoided in some situations by using multiple redundant markers arranged in a known configuration (see, for example, the virtual shared whiteboard in Figure 3.1); thus if any of the markers are detected, the locations of the remaining ones may be inferred [Kato and Billinghurst 1999].

3.2.2 Tracking method

The markers used in ARToolKit consist of simple black and white designs[4]. While sample markers are provided, the application developer is free to create their own custom patterns. The system is able to recognise multiple markers simultaneously and to distinguish among different marker patterns.

Consider the design of the ARToolKit markers: the main feature is a black square on a white background. This distinctive shape allows the system to quickly identify potential markers using the following algorithm, illustrated in Figure 3.4:

1. Threshold the image.
2. Label connected components.
3. Trace the outline of each labeled region.
4. Fit a square to the traced outline, eliminating non-square regions.

[4]Greyscale and colour markers are also supported, though rarely used.

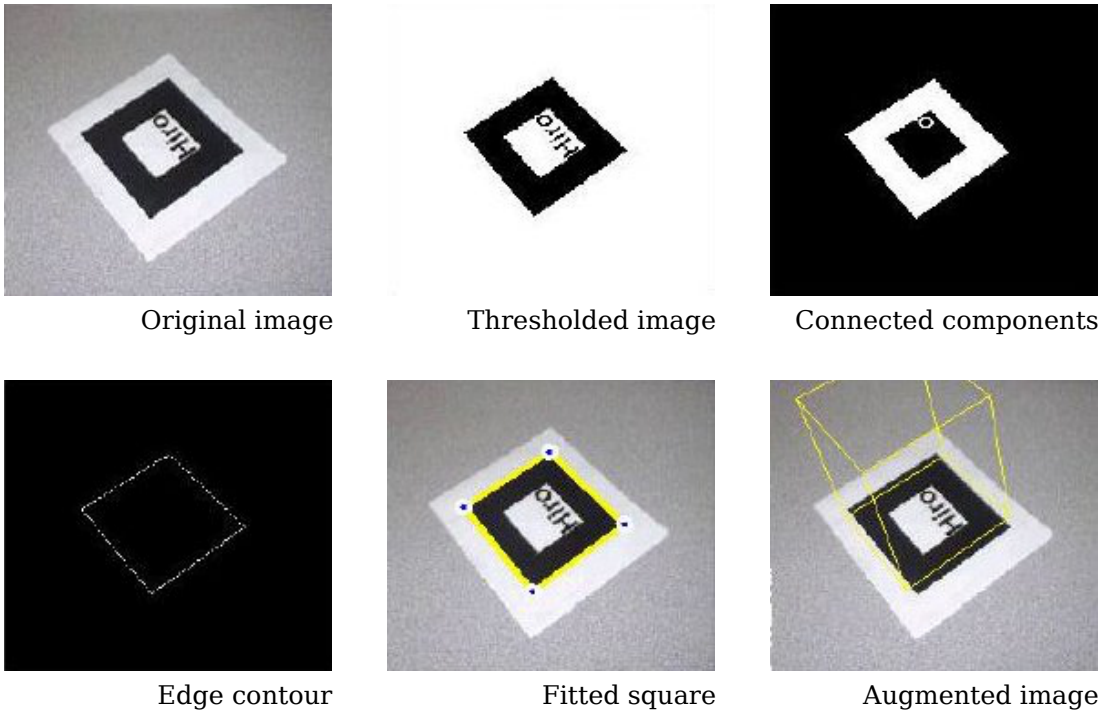


Figure 3.4: Outline of the computer vision algorithm implemented by the ARToolKit, showing the original image and the result after each major processing step. (Adapted from http://www.hitl.washington.edu/artoolkit/documentation/vision.htm)

In the ARToolKit implementation of the above process, simplicity and speed are favoured over accuracy. A fixed image threshold is used, meaning markers can be lost in dim or bright scenes. Also, the detection routine requires an unobstructed view of the marker border, making it highly susceptible to occlusion due to intervening objects or viewport clipping.

The second important feature of each marker is a unique identifying pattern that serves to differentiate it from other markers and from non-marker objects. For each potential marker, the pattern is extracted and compared to a database of known patterns. The best match is returned along with a 'confidence factor' indicating the strength of the match.

The matching technique used by ARToolKit is based on a simple cross-correlation of low-resolution pattern images. This method is not entirely reliable and, as previously mentioned, can lead to markers being mis-identified with a high confidence factor [Fiala 2005a, Zhang et al 2002].

Processing an image with ARToolKit thus results in the following information:

- A list of potential markers, including the location of the four corners of each (in image coordinates).
- For each detected marker, a pattern selected from the pattern database.

In general, the application developer will require the three-dimensional position and orientation of the marker relative to the real camera. ARToolKit can calculate this from the above information, assuming a fixed marker size.

3.2.3 Performance evaluation

On a modern system with hardware-accelerated graphics, high framerates are readily achievable. Table 3.1 shows some typical values for various demo applications. These results were obtained by the author by letting each program run for 60 seconds while moving the camera about a scene in which multiple markers were visible. The test machine was a dual-core 3.2GHz Pentium 4 running the Linux kernel.

Each demo program follows a similar process. It receives a video frame from the video back-end (in this case, GStreamer), calls ARToolKit functions to detect markers and calculate marker poses and then makes OpenGL calls to display the video frame along with any virtual objects. The following demos were selected for testing:

- simpleTest—A very simple demo that draws a cube on top of the marker.
- relationTest—This demo tracks two different markers and calculates their relative displacement in three dimensions. It draws a sphere on one and a cone on the other.
- rangeTest—This demo tracks the marker and calculates its distance from the camera; it also draws a 'teapot' object on top of the marker.
- paddleDemo—This demo uses the six-marker board (featured in figure 3.1) as a virtual ground drawn as a red grid upon which green spheres rest. Another marker becomes a paddle for moving the spheres about.
- collideTest—This demo tracks two different markers and determines if they are touching one another. It draws a simple object on each marker: a sphere if they collide, a cube otherwise.

In each test, the input data was sixty seconds of live video from a handheld webcam, where the camera was moved about to simulate a typical usage scenario. The results obtained by this method are variable because the actual processing time is dependent on factors such as how many markers are visible in the scene and how large each marker appears. However, they do indicate which functions are the most time-consuming under normal use. Each program was tested twice; once with hardware-accelerated graphics rendering, the second time without.

The average framerate for each test is reported in Table 3.1, where the difference in performance when using hardware and software rendering is apparent. The speedup provided by accelerated graphics was over fifty times for the simpleTest, relationTest, and

Application	Frames per second	
	Accel.	Non-accel.
simpleTest	201	8.79
relationTest	172	7.91
rangeTest	150	2.51
paddleDemo	126	8.12
collideTest	135	2.58

Table 3.1: Typical framerates for selected ARToolKit demo applications, both with and without hardware acceleration. Note that the camera framerate is 30 fps, so in the accelerated case each frame is processed more than once.

paddleDemo programs, but only around twenty times for the rangeTest, and collideTest demo. This discrepancy is explained by the different resolutions of these programs (unchanged from their default values): simpleTest, relationTest, and paddleDemo used the camera's default resolution of 640×480 pixels, while the other two upscaled the frame to twice this size. In these simple examples, the system is so fast (with hardware acceleration enabled, at least) that it processes each camera frame multiple times[5].

Timing statistics were provided by the `gprof` profiling tool. Using this tool, detailed timing information was available on each of the ARToolKit functions. This information was used to produce the chart in Figure 3.5, showing the relative time taken by the different parts of the marker registration process. The graphs of the first three programs (simpleTest, relationTest, and rangeTest) consistently show around 90% of the processing time going to the arDetectMarker function, with the remaining 10% going to arGetTransMat. The remaining two programs use variants of these two functions, and show a slightly different ratio between the two operations.

arDetectMarker picks potential markers from a frame and is called exactly once each time the image is updated. In contrast, arGetTransMat (which calculates the position and orientation of a marker) is executed for every marker detected, so may be called more than once, or not at all, in a single update. Looking at the second level of function calls, it is apparent that the most time-consuming operation is arLabeling, which labels the connected components in the image. The paddleDemo, and collideTest programs show different results due to their use of variants of these routines (for example, arDetectMarkerLite instead of arDetectMarker in paddleDemo); still, more time is spent on labeling connected components than on all other operations combined.

The timing statistics provided by `gprof` does not include operations such as graphics drawing and video frame retrieval as these are not visible to the profiler. Thus the results shown in Figure 3.5 represent only part of the program activity each frame. Figure 3.6 presents further information about each test. The darker-shaded parts of the bars show the amount of time per frame accounted for by the profiled functions.

[5]The test applications use the GStreamer video back-end which, by default, provides a frame whenever a request is made. Other video back-ends will block until a new frame is available.

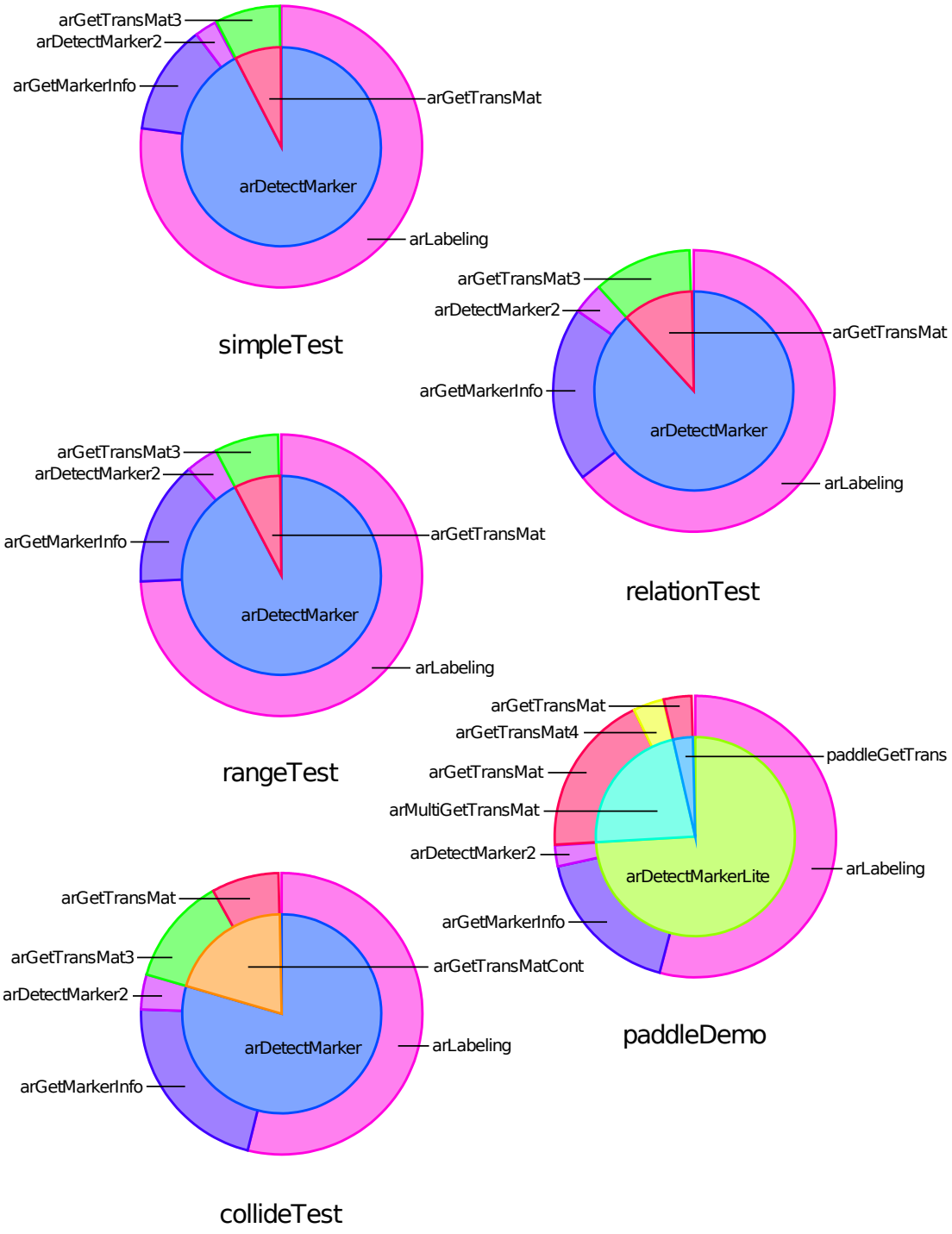


Figure 3.5: Relative time spent on different ARToolKit calls, for each of the five demo programs. The inner circle shows the top-level function calls, while the outer ring shows the sub-functions called by that function. The results shown are those for the hardware-accelerated case.

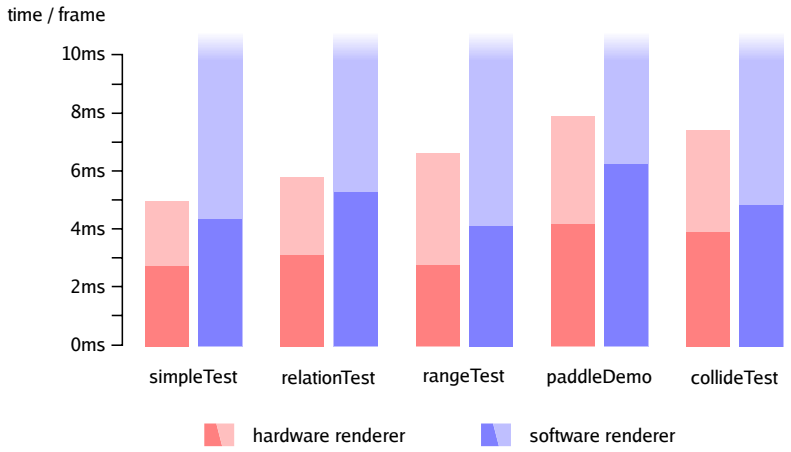


Figure 3.6: Average time taken per frame for each of the selected ARToolKit demo programs. Note that the total time taken by the software renderer was around 200 ms per frame, thus it cannot be represented on this graph. The darker part of each bar indicates the proportion of the test accounted for in the profiler results.

The remainder of each bar (the lighter-shaded part) indicates operations that were not visible to the profiler. The total height of each bar thus represents the total time taken per frame; note that the software rendering tests, at 100 ms to 200 ms per frame, do not fit on the graph.

The high framerates yielded by these tests reinforce the results of a 2002 study [Zhang et al 2002] that found ARToolKit compared favourably with other marker systems in terms of speed. The same study confirmed the accuracy of the corner detection method, finding an average error of less than two pixels (though it should be noted that the other systems managed even better).

In an evenly lit environment, the 3D position and orientation values that ARToolKit supplies are generally accurate and show negligible jitter. A major problem with the system, though, is that it is prone to losing track of the markers. Some reasons for tracking failure are expected, such as the case where the marker is obscured by motion blur, or is viewed at an extreme angle. On the other hand, due to the detection process it employs, the ARToolKit is particularly sensitive to the problem of occlusion—if the marker border is interrupted by an intervening object, the frame of the camera, or even a strong shadow, it will fail to recognise the marker.

3.3 BAZAR

BazAR enables template-based AR tracking using two-dimensional images. It is an implementation of the object detection and registration technique described in the paper “Keypoint Recognition using Randomized Trees” [Lepetit and Fua 2006]. The library is written and distributed by the Computer Vision Lab, EPFL[6].

[6] http://cvlab.epfl.ch/

Figure 3.7: BazAR object registration by matching keypoints. Interest points in the object image (top half of figure) are sought out in the scene image (bottom half of figure). Given enough corresponding points, the object pose can be calculated. (Image from http://cvlab.epfl.ch/software/bazar/.)

Using BazAR, an application can determine whether or not a specific object appears in a scene and, if it does, calculate its position and orientation. The object must be planar (or nearly so) and the system must be trained to recognise it ahead of time. Supporting functions (such as video stream handling) are supplied by the OpenCV[7] library.

BazAR works by matching keypoints on the object to keypoints in the scene, as illustrated by figure 3.7. This approach has two advantages over that of ARToolKit: first, the system can detect arbitrary objects and not just markers; second, the object can be registered even if some keypoints cannot be matched, making it robust to effects such as occlusion.

The source code is licensed under the GPL and can be downloaded from the BazAR homepage at http://cvlab.epfl.ch/software/bazar/. Both Unix/Linux and Windows versions are available.

3.3.1 Model-based systems

A less-obtrusive alternative to marker-based systems employs a model of either the local environment or specific objects of interest as a basis for tracking. These models are constructed ahead of time and take a variety of forms, such as a spatial arrangement of basic geometric features provided by the user [Comport et al 2003, Comport et al 2006] or a set of image features learned by the system during a dedicated training phase [Genc et al 2002, Skrypnyk and Lowe 2004]. These features are reconciled with those the system observes in the scene, thus it orients itself relative to the modelled object.

[7] http://opencv.willowgarage.com/

Figure 3.8: The Virtual Visual Servoing framework [Comport et al 2003] being used to track a simple model consisting of two straight lines and the 2D projection of a cylinder. The left image demonstrates the ability of the system to detect these basic geometric features in spite of occluding objects. In the right image, this tracking information is used to augment the scene with virtual objects (a penguin and a columnar object).

Early model-tracking frameworks made use of edge features, as these are both computationally efficient and relatively easy to detect [Lepetit and Fua 2005]. The model would be formed from basic geometric primitives and was constructed by the user in advance. Figure 3.8 shows an application of this technique to AR, tracking a simple model consisting of a cylinder and two straight lines [Comport et al 2003]. As is observable in the figure, this system produces accurate tracking information even when parts of the model are occluded.

While edge-based methods are straightforward to implement, they exploit relatively little of the available image information and for this reason interest-point-based methods may be used instead [Lepetit and Fua 2005]. In this case, the model is constructed by extracting common feature points from a set of images of the object to be augmented. An example of this approach is the system described in Skrypnyk and Lowe [2004], which makes use of *scale-invariant feature transform* (SIFT) features [Lowe 1999]. The authors claim accuracy comparable to ARToolKit; furthermore, the process is robust to visual interference, as demonstrated by the examples in Figure 3.9.

BazAR uses a variant of the model-based approach that works with 2D templates instead of 3D models. The method employed by BazAR is an implementation of the technique described in Lepetit and Fua [2006]. This technique also makes use of interest points, in this case to train a classifier to detect an object. Although BazAR can only handle rigid planar objects, the method can be applied to non-planar and even deformable objects as well. Features are identified as a result of a sequence of binary tests comparing the grey-levels of randomly selected pixels in the region of the feature. The authors claim that this approach is more computationally effective than the SIFT method, yielding a framerate of 25 fps on a 2.8 GHz PC (compared to 4–5 fps on a

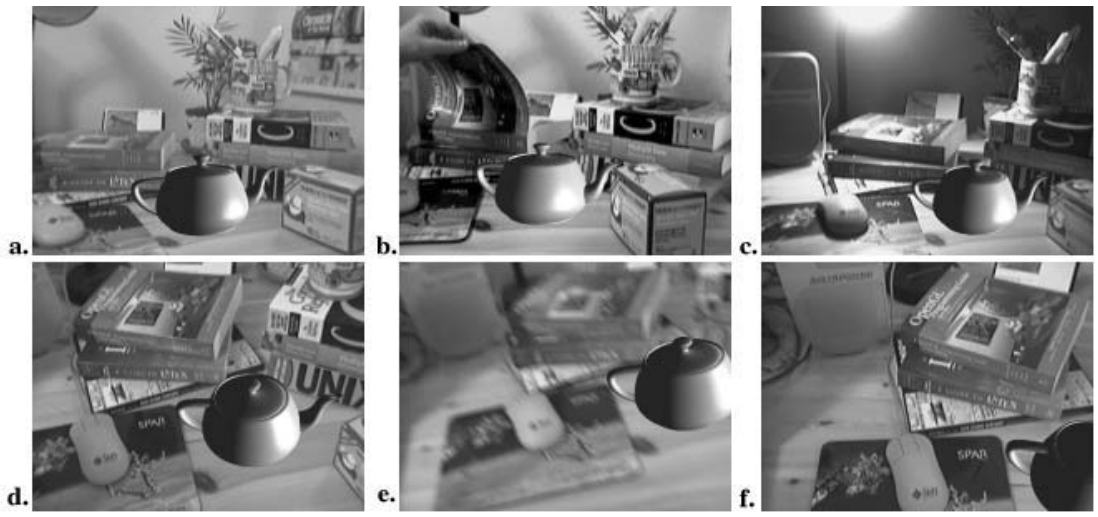


Figure 3.9: SIFT-based scene modelling and tracking framework, from Skrypnyk and Lowe [2004]. A virtual object, in this case a teapot, is positioned in the modelled scene (a). Tracking is successfully maintained in spite of moving objects (b), lighting changes (c), viewpoint shifts (d), motion blur (e), and an incomplete view of the scene (f).

less-powerful 1.8 GHz machine for the SIFT-based method [Skrypnyk and Lowe 2004]), while being more robust to perspective distortion. However, they note that the cost of this robustness is a training phase that the SIFT method does not require.

Despite the computational complexity of these methods, versions have been made to run on platforms with limited resources. Wagner et al. have developed modified versions of both the Ferns[8] [Ozuysal et al 2007] and SIFT-based methods for mobile devices [Wagner et al 2008a]. The authors find the performance of both techniques to be comparable given an equal level of CPU performance and memory, though they note that "At the lower end, SIFT may be more stable than Ferns, while Ferns may be able to outperform SIFT given an increased amount of memory and CPU bandwidth currently not afforded by phones." The average runtime per frame was around 60 ms when running on a 330 MHz TI OMAP 2420 processor, corresponding to a potential framerate of 15 fps.

A different algorithm was implemented on the same processor [Chen et al 2007]. A reduction in performance by an average factor of 22 was observed, compared to a 1.8 GHz dual-core desktop machine. The authors suggest that performance could be improved by vectorising parts of the code to take advantage of the built-in floating point SIMD unit.

[8]Ferns is a variation of the classification technique described previously, involving a simplified decision algorithm.

3.3.2 Tracking method

With BazAR, the approach taken to object detection is to treat it as a classification problem. Keypoints are extracted from the incoming video frame and a classifier is employed to determine which object keypoint, if any, each detected keypoint corresponds to. Having established a set of matching keypoints, the object's pose is calculated using the RANSAC [Fischler and Bolles 1981] model-fitting algorithm.

The keypoint classifier is trained from a collection of images showing the object to be detected from multiple different viewpoints. In the planar object case, the user need provide only a single (frontal) image of the object; the system will automatically generate the required views. Classifier training is carried out in an offline phase.

Keypoint classification consists of a series of tests comparing the intensity of pixels in the region of the point of interest. Multiple binary decision trees are randomly formed from these tests. The result of each tree is combined to produce the final classification.

3.3.3 Performance evaluation

The "augment3d" sample program included with BazAR augments a video stream with a virtual marker showing the position of a tracked object. This program ran at an average of 10 frames per second on the test machine[9]. The time taken and disk space used to train the classifier varied according to the number of keypoints found: an object model with only 23 keypoints took 5 seconds to train and 1.1 MB of space, while one with 342 keypoints took 41 seconds and 16 MB.

The reliability of the detection algorithm and the accuracy of the estimated pose depend heavily on the number of identifiable keypoints the object provides. In general, the following behaviour was observed:

- 'Good' objects with at least 200 keypoints could be detected fairly consistently, though the accuracy of the calculated pose was variable. With a clear view of the object the pose would closely match the observed orientation. Reliable pose calculation was not possible with less-ideal views.

- Objects with 100–200 keypoints could be detected given a clear view of the object, but pose estimation was unreliable.

- 'Poor' objects with less than 100 keypoints could not be detected with any consistency. Simple graphical objects such as logos and ARToolKit-style markers tended to fall into this category.

[9]The results reported in this section are based on a series of tests carried out by the author. Version 1.3.1 of the library was used, running on a dual-core 3.2 GHz Linux machine.

A "clear view" in this case indicates the object appears front-on to the camera and takes up a significant portion (around 30%) of the frame. Motion blur and poor illumination also noticeably degrade the ability of the system to accurately register objects. It does, however, exhibit strong robustness to occlusion.

3.4 PTAM

The parallel tracking and mapping (PTAM) technique is an environment tracking system designed to support AR applications. Many AR applications are built around a desktop, whiteboard, or other flat work surface. This system, by Georg Klein [Klein and Murray 2007], offers a way to track this surface without the need for markers or a predetermined model.

A cross-platform demo application of PTAM is available online from the author's homepage[10]. The code for this program is provided as a reference implementation of the technique but is not intended for use in real applications (and in fact its licence specifically prohibits its commercial use).

3.4.1 Extendible tracking techniques

With the various marker-based and model-based techniques described previously, a common factor has been the tracking strategy employed. The marker or model is detected afresh in every frame and it is assumed that the detection routine is reliable enough to support this. Robustness may be improved by modifying this tracking strategy to consider the progression of the object (say, a marker) across multiple frames. In this situation, a motion model is applied to the marker, fed by the results of the detector. The motion model thus provides an estimate of the marker pose in every frame, even when the detector fails [Koller et al 1997].

A variation of this technique tracks naturally-occurring features in the environment, instead of the artificial features provided by a marker. The natural features are used to form an estimate of the camera motion, though without an external frame of reference this technique is limited to 2D [Neumann and You 1999]. Markers may provide a suitable frame of reference, allowing full 3D motion tracking as shown in Figure 3.10.

This technique may be further enhanced by using the tracked features to construct a model of the scene over time, similar to the methods described above except that the model is not fixed at the start. Such a system is implemented in Subbarao et al [2005], using markers to provide an initial frame of reference. Each frame is checked to see if the markers are visible; if so, the marker pose is used to correct the model, otherwise scene features are used instead. In either case, any new features detected are integrated into the model and outliers are removed.

[10] http://www.robots.ox.ac.uk/~gk/PTAM/

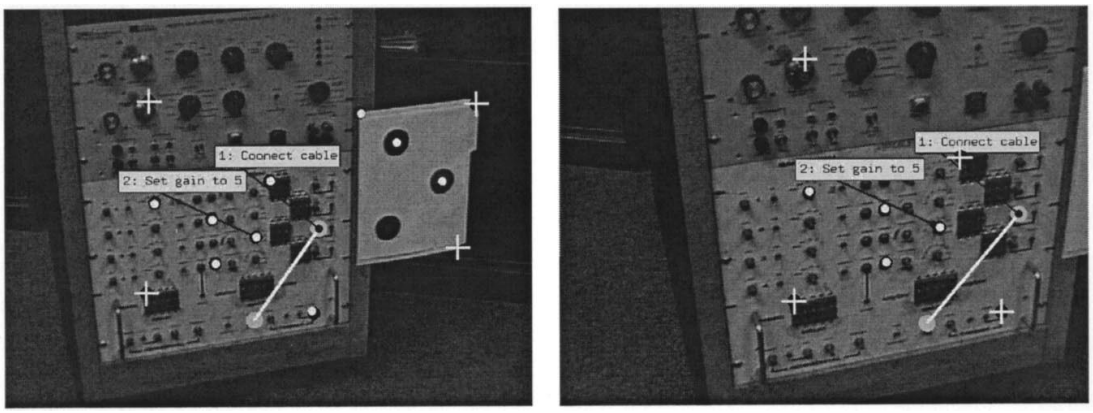


Figure 3.10: Two shots of the natural feature tracking system proposed in Neumann and You [1999], in which an equipment rack is annotated with informational tags. The trio of black circles in the left image are markers, required to establish an initial frame of reference for the tracked features.

This system is an application of the structure from motion (SFM) technique for simultaneous recovery of scene structure and camera motion. An alternative approach, drawn from the field of robotics, is simultaneous localisation and mapping (SLAM), which refers to the problem of determining the position of a robot in an unknown environment. More specifically, it is the process by which a mobile robot incrementally forms a map of its surroundings and, at the same time, uses this map to estimate its location [Durrant Whyte and Bailey 2006]. The key to this process is the formation of *landmarks*, points in 3D space relative to which the robot orients itself. The use of such permanent features for tracking makes SLAM-based systems less susceptible to drift than those utilising SFM [Davison et al 2003]. However, traditional SFM techniques have the advantages of robustness and high-performance [Bleser et al 2006].

The Parallel Tracking and Mapping (PTAM) method is based on SLAM techniques, with the notable feature that the twin tasks of environment mapping and motion tracking are carried out in two separate threads that run simultaneously. The method assumes the presence of a dominant ground plane in the scene to be augmented (such as the desktop in Figure 3.11), thus providing a frame of reference in the absence of markers.

3.4.2 Tracking method

The PTAM technique is built on the two fundamental and interdependent operations of camera tracking and environment matching. As new data (i.e., a new camera frame) is fed into the system the tracking stage updates the estimated position and orientation of the camera while the mapping algorithm incrementally forms a map of the surrounding environment. These correspond to the localisation and mapping tasks of the SLAM problem.

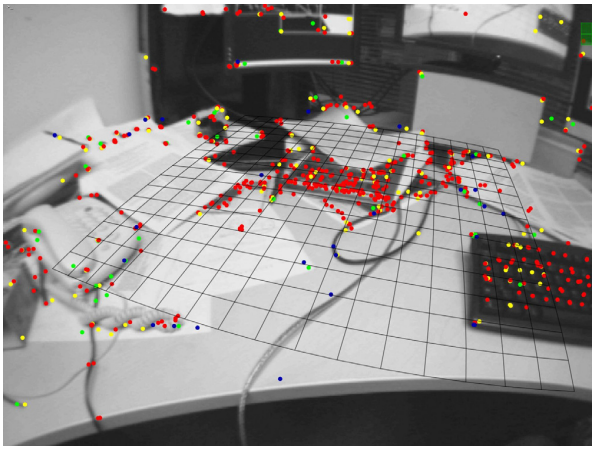


Figure 3.11: The SLAM-based parallel tracking and mapping system being used to track a typical desktop workspace. The successfully-tracked feature points are picked out with dots, and used to estimate a ground plane (shown as a grid). (From "Parallel tracking and mapping for small workspaces" [Klein and Murray 2007].)

The environment map consists of all the keypoints that have been observed in previous frames, along with their calculated positions in 3D. It is not updated with every new frame from the camera but is instead formed from a limited set of key frames. Periodically, a new keyframe will be added to this set and the map recomputed. The camera pose is updated each frame and is determined by refinement of the expected pose using keypoints from the environment map.

PTAM detects the dominant planar region in the initial environment map and registers it as the surface for augmentation. As the work surface has a fixed location within the map, its position relative to the camera is known regardless of whether it is clearly visible, partially obscured or even entirely out of frame. However, in order for this mapping technique to be valid the environment must be mostly static.

3.4.3 Performance evaluation

The demo application provided with the source code applies the PTAM process to a live video feed and presents the resulting augmented image stream. Superimposed on the image stream are the tracked keypoints as well as the location of the ground plane, assuming it is in view; these features may be observed in the screen capture of Figure 3.11. It runs at full camera resolution and framerate (640×480 at 60 fps) on a dual-core 2.8GHz machine.

When the area visible to the camera is well-represented in the environment map, the tracking results are stable and free of jitter. However, it takes time to accumulate sufficient keypoints to accurately register a new part of the scene. This limits the speed at which the camera can be moved about an unknown environment if accurate tracking is to be maintained.

3.5 SUMMARY

The three tracking techniques described in this chapter are representative of the various approaches to the registration problem in vision-based augmented reality. ARToolKit implements a simple marker tracking algorithm that provides reliable results when the marker is clearly visible, though has poor robustness to interference. The BazAR template-matching process can theoretically track arbitrary images; in practice, however, only some images provide sufficient features for reliable tracking and even 'good' images cannot always be accurately registered. PTAM makes use of techniques drawn from robotics research to track the naturally-occurring features in the environment, but cannot deal with sharp changes in viewpoint.

While ARToolKit and PTAM were able to operate at the full camera framerate (60 frames per second), the BazAR application managed only 10 frames per second. Processing speed is an important consideration for the project described in this thesis as the processing power available in an embedded system is typically less than that for a desktop computer. Chapter 6 discusses the development of an AR application based on the ARToolKit tracking routines and presents performance figures using the embedded device developed for this project.

Chapter 4

SYSTEM ARCHITECTURE

4.1 INTRODUCTION

This chapter explores the different processing technologies that are used in embedded systems and examines their use in embedded computer vision and augmented reality applications. These processing technologies range from standard general-purpose processors to more specialised devices such as DSPs and FPGAs. Another issue that the system designer must consider is whether or not a single processing element will be able to provide sufficient computational power for the application.

These issues are considered in Section 4.2, with reference to existing computer-vision and multimedia devices. The limited number of cases where embedded technology has been employed in augmented reality applications are surveyed in Section 4.3. Section 4.4 presents two possible designs for an embedded augmented reality platform; one based on a single DSP, the other using a multiprocessor system-on-chip.

4.2 APPROACHES TO EMBEDDED SYSTEMS DESIGN

The study of embedded platforms for augmented reality applications is extremely limited, particularly for embedded computer vision for AR. Fortunately, there is a well-established body of research on more general embedded computer vision architectures. Since one of the two major tasks of a vision-based AR system is the tracking algorithm (the other being the rendering of virtual objects), it is relevant to examine the available platforms for embedded computer vision.

Embedded computer vision platforms come in a variety of forms, depending on which of the many available processor types are employed. Specialised processing elements, including digital signal processors (DSPs) and field-programmable gate arrays (FPGAs), allow a device to be tailored to a specific application [Baumgartner et al 2009].

An alternative approach is to make use of a general-purpose processor (possibly backed up by one or more special-purpose processors) to create a system that can handle

a range of tasks. This is the approach favoured by developers of modern mobile phones and has given rise to a variety of multimedia-oriented "system-on-chip" processors such as Texas Instruments' Open Multimedia Applications Platform (OMAP) [Chaoui et al 2001].

4.2.1 Embedded computer vision platforms

Embedded hardware for computer vision applications comes in many different forms. A wide variety of processing architectures exist, including general-purpose processors, special-purpose processors (such as digital signal processors, or DSPs, and graphics processing units), and customised hardware (FPGAs and ASICs) [Kölsch and Butner 2009].

Different architectures have different strengths and weaknesses and the selection of one or the other will be guided by the design constraints of the application. Low-level algorithms with a high degree of parallelism are a good fit for FPGAs, while higher-level algorithms involving branches and loops are better suited to DSPs [Baumgartner et al 2009]. Weaknesses in one device may be compensated for by pairing it with a different type; for example, the combination of a DSP and FPGA to enable high-performance vision systems [Fürtler et al 2007].

A developer has a number of options for forming these processing elements into a complete device. The most straightforward approach is to use standard PC hardware available in an embedded form-factor, such as that specified by the PC/104[1] standard. This base platform can be extended with auxiliary boards providing additional processing resources including DSPs and FPGAs. One drawback of such systems is the power they consume. A typical power budget may be in the region of 50 W [Kölsch and Butner 2009]. On the other hand, the use of such hardware allows an application to be developed on a desktop PC without the difficulties associated with emulating or porting to a different architecture.

While PC-based systems tend to be loosely-integrated, *smart cameras* take the opposite approach. In devices of this type, a processing element is tightly coupled with the image sensor, forming a single unit capable of performing sophisticated video analysis in real time [Wolf et al 2002]. This is the approach chosen by the developers of the "SmartCam" system [Caarls et al 2002], with its array of SIMD processing elements and optional instruction-level-parallel processor. More recent devices have utilised less-exotic processors such as a microcontroller [Rowe et al 2007] or DSP [Xu et al 2008].

Application development for these devices can be complicated by the lack of a memory-management unit (MMU) on such processors. The MMU handles virtual memory, required by regular operating systems for multitasking. A non-standard OS

[1] http://www.pc104.org/

must be used instead (uClinux[2], for example), or the application developer may choose to forego an OS entirely.

In applications where no single processing element can provide sufficient computing power, one solution is to harness multiple processors together in a single system. This has lead to the development of the multiprocessor system-on-chip (SoC), combining the processors along with elements such as embedded DRAM, flash memory, application specific hardware accelerators and RF components into a single package [Ravikumar 2004]. Advantages of this approach include the potential openness of the software infrastructure of generic microprocessors and the efficiency of synthesized hardware [Roza 2001].

4.2.2 Mobile multimedia devices

The SoC-type device has found widespread applicability in the mobile communications industry. Modern mobile phones offer a great deal of power and functionality in a compact and convenient form factor. For example, a typical high-end consumer phone from 2009 is the Nokia N900[3], with the following features:

- 600 MHz ARM processor with 256 MB RAM
- Full multitasking operating system (Linux-based)
- 800×480 touch-screen LCD
- 3D graphics accelerator
- 32 GB internal storage (extendable with microSD card)
- 3.5G and WLAN wireless communications
- 5-megapixel digital camera
- Built-in GPS

The type of SoC that allows this device (along with many others currently on the market) to offer the above features is known as an "applications processor". In the N900, the processor is the OMAP3430 from Texas Instruments[4], but similar chips are available from Qualcomm[5] and Samsung[6]. The feature set of each chip varies from manufacturer to manufacturer, and from model to model, but the following characteristics are common to all such devices:

[2] http://www.uclinux.org/
[3] http://maemo.nokia.com/n900/
[4] http://focus.ti.com/general/docs/wtbu/wtbugencontent.tsp?templateId=6123&navigationId=11988&contentId=4638
[5] http://www.qualcomm.com/products_services/chipsets/snapdragon.html
[6] http://www.samsung.com/global/business/semiconductor/products/mobilesoc/Products_ApplicationProcessor.html

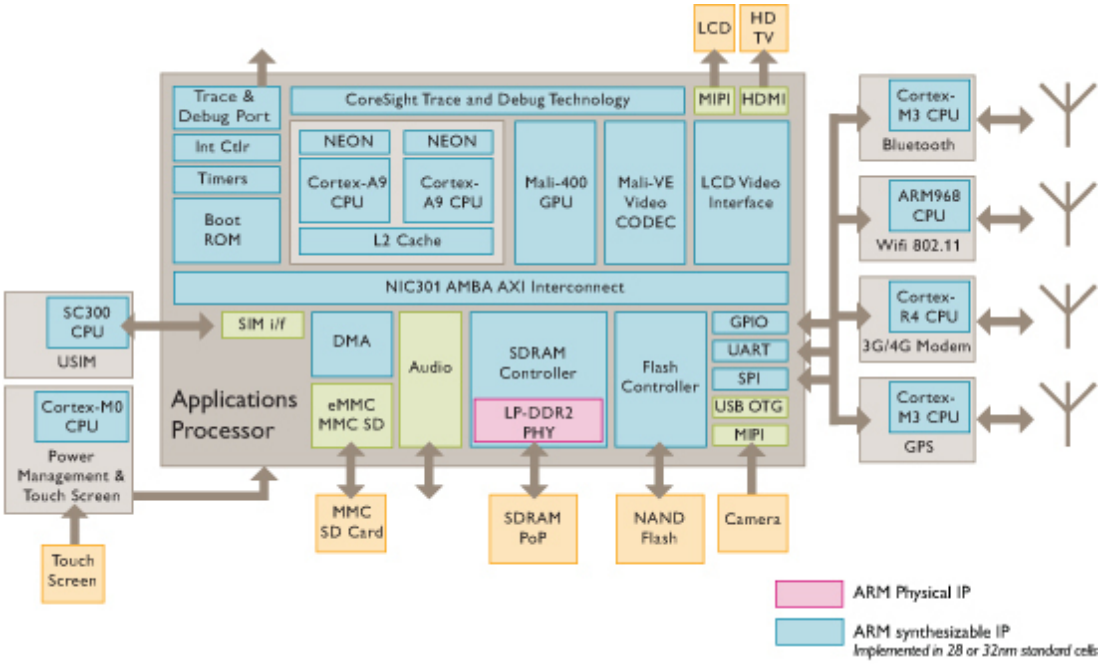


Figure 4.1: Optimized ARM smartphone block diagram. The "applications processor" in this diagram is a hypothetical design supplied by ARM. It is typical of the chips offered by Texas Instruments, Qualcomm, and other manufacturers in terms of processing hardware and on-board peripheral interfaces. Surrounding the applications processor are additional devices and subsystems that may be expected on a modern smartphone. (From http://www.arm.com/markets/mobile/smartphones.php.)

- One or more ARM processor cores.
- Optional extra processing elements, such as a DSP or graphics accelerator.
- Memory control circuitry for interfacing external RAM.
- A wide set of common peripheral interfaces.
- A Direct Memory Access (DMA) controller for efficient data transfer.

The prototypical applications processor shown in Figure 4.1 presents a typical feature set for this type of chip. This unit includes dual ARM Cortex-A9 CPUs, hardware graphics and video accelerators, and controllers for a variety of peripherals. SDRAM takes the form of an additional chip mounted on the main processor in PoP (package-on-package) configuration. It is interesting to note the correspondence between the functional blocks in the diagram and the feature set of the N900. For example, the built-in LCD, camera and memory card interfaces (all of which are present in the OMAP3430) are put to use in the Nokia device.

The name "applications processor" derives from Cortex-A[7], the name given to the series of ARM processors used in many of these devices. An important feature

[7] http://www.arm.com/products/processors/cortex-a/index.php

of these processors is their ability to run a full multi-tasking operating system. The N900, for example, runs the Linux-based Maemo OS[8], while many more use the (also Linux-based) Android OS[9] developed by Google.

4.3 EMBEDDED AUGMENTED REALITY

In the discussion on the history of augmented reality in Chapter 2, the point was made that, until mobile phones gained advanced media-handling capabilities, little interest was shown in augmented reality on embedded platforms. Nevertheless, there have been a few devices constructed for the purpose of enabling various AR systems. Based on DSPs or FPGAs, they concentrated on the (vision-based) tracking stage of the AR process and tended to neglect the generation of virtual objects.

As mobile phones became more powerful, a number of groups around the world began investigating them as potential platforms for AR. By 2007, the more advanced phones were capable of hosting AR applications with a similar level of complexity to their desktop counterparts, with both marker-based and markerless tracking schemes on offer. The emergence of mobile AR was discussed in Section 2.4.

4.3.1 Custom embedded AR platforms

Instances of embedded AR platforms are rare in the history of AR research, with development instead focused on PC-style systems (as described in Section 2.4). The earliest known example of an embedded AR system, described in Uenohara and Kanade [1995], used multiple TMS320C40 DSPs to implement object recognition and feature-point detection algorithms, in addition to generating the virtual object overlay. This system operated in real-time at 30 frames per second (the framerate of the camera used).

The first use of an FPGA for augmented reality was documented in Luk et al [1999]. This system could track objects and generate the augmented display but was designed to work in conjunction with a regular PC, not as a stand-alone device. DSPs were again employed in Naimark and Foxlin [2002], this time to track circular markers, and in Smith et al [2005] a portable laptop-based AR system was presented that used an FPGA to track the user's hand.

Most recently, Guimarães et al [2007] put forward another AR system that used an FPGA for vision-based tracking. Unlike the other FPGA-based systems mentioned previously, this one was designed as a stand-alone unit, requiring no supporting computer. Figure 4.2 shows the important processing cores and datapaths in this system. An image sensor, driven by the *I2C image sensor control*, provides a video stream.

[8] http://maemo.org/
[9] http://android.com/

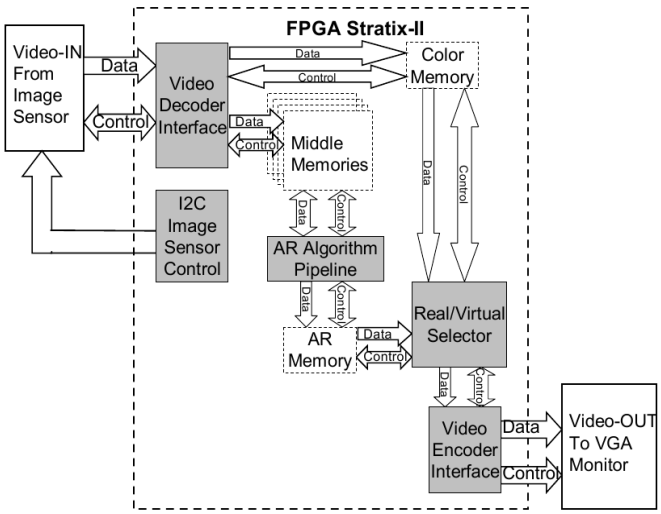


Figure 4.2: A block diagram of the FPGA-based AR platform implemented by Guimarães et al, with an image sensor providing input, a VGA monitor to display the output, and a number of functional cores hosted by the FPGA. (From Guimarães et al [2007].)

Image data is consumed by the *video decoder interface* and produced by the *video encoder interface*. Two parallel datapaths connect these two blocks, one passing the data through unmodified, the other leading to the *AR algorithm pipeline*. The two streams are combined by the *real/virtual selector* before being output to a monitor.

A variety of basic image operations (such as edge detection and generic convolution) were implemented. These operational units could be assembled in different ways to form the AR algorithm pipeline, depending on the application. Performance comparisons were carried out, showing much improved efficiency in terms of clock-cycles-per-operation compared to a desktop computer. In absolute terms, however, certain operations took longer to complete due to the FPGA's slower clock (100 MHz vs. 2 GHz for the PC).

4.3.2 AR on mobile media devices

In contrast to the scarcity of research into custom embedded platforms, the study of mobile AR is one of the most important emerging research areas in the field [Zhou et al 2008]. Researchers in this area benefit from the rapid pace at which mobile phones and similar devices are evolving. A high-resolution camera and a 3D graphics accelerator are now standard on many phones, making them appealing platforms for vision-based augmented reality applications [Wagner and Schmalstieg 2009a, Wagner and Schmalstieg 2009b].

As was the case in the non-mobile domain, the first vision-based tracking schemes to be implemented on mobile devices made use of artificial features in the form of

markers[10]. Early attempts to develop mobile marker-tracking systems were made using PDAs [Wagner and Schmalstieg 2003] and smartphones [Henrysson and Ollila 2004, Möhring et al 2004]. These devices proved to be underpowered, yielding less than ten frames per second.

By 2007, mobile phones employing multimedia-oriented applications processors (see Section 4.2.2) were available. These devices proved to be suitable platforms not only for marker-based systems [Wagner and Schmalstieg 2007] but natural feature trackers as well [Chen et al 2007].

The Studierstube ES library was specifically designed for such devices [Schmalstieg and Wagner 2007]. This AR library provided an implementation of a marker-based tracking scheme (similar to ARToolKit and ARTag; see Section 3.2.1). The cited paper presents a series of benchmarks that were undertaken on five different applications-processor-based phones. Framerates were found to be highly dependent on both the clock frequency and the complexity of the 3D object model, with one 500 MHz device delivering around 25 fps while rendering a medium-scale model. Another (330 MHz) device, this one equipped with a GPU, achieved around 40 fps when rendering the same model.

Current-generation applications processors are clocked at higher rates and are more likely to include on-board graphics hardware. For example, the previously-mentioned OMAP3430 includes a POWERVR SGX 2D/3D graphics accelerator (along with a C64x+ DSP) and can be clocked at 600 MHz.

4.4 PROPOSED DESIGN

The device that is to be built must possess three essential features:

- A camera, to provide the video stream.
- A display, to present the result.
- A processing unit, including a fast enough processor and sufficient memory to execute the AR application.

In Section 4.2 a variety of processor types were described. The different approaches offer different advantages and disadvantages in terms of flexibility and power.

In this section, two designs are presented that could potentially satisfy the requirements of this project. The first is a type of smart camera (as defined in Section 4.2.1); the second is more similar to a multimedia-capable smart phone.

[10]Not considered here are systems that offload the tracking computation to a remote machine (such as the one described in Gausemeier et al [2003]).

4.4.1 AR smart camera

Section 4.2.1 presented an overview of the options a designer of embedded systems has when faced with the task of building an embedded computer platform. Considering these options in the context of the present task, the first point to note is that the simplest approach, that of using a small PC-compatible board, is unsuitable. The bulk of such a device would be unacceptable, particularly considering the mass of batteries that would be required to power it.

The alternative approach uses specialised processing elements such as DSPs and FPGAs, selected on the basis of the specific requirements of the intended application. As covered in Section 4.3.1, it is this approach that embedded systems developers in the AR field have largely followed. Such devices can be expected to be low-powered and lightweight.

One design considered for this project was to create a smart-camera-style device using a Blackfin processor[11]. This processor architecture is a hybrid DSP/microcontroller created by Analog Devices, Inc. and has been used in previous projects here at the University of Canterbury (e.g., Loten and Green [2008]). Though lacking an MMU, the uClinux variant of the Linux kernel is supported on this device[12] and a GCC-based toolchain is available[13].

Blackfin processors have a number of features that make them appealing to computer vision researchers, such as a large cache size, multiple video streaming capability, and a symmetric dual-core design (in the case of the BF561, at least) [Loten and Green 2008]. The availability of kits such as the Surveyor SRV-1 Blackfin camera[14] demonstrates their suitability for this type of workload.

Less certain, however, is their ability to handle the other major task of a typical AR system; that of rendering virtual objects in 3D. As Schmalstieg and Wagner [2007] showed, a hardware renderer is necessary for reasonable performance. It is worth noting that of the two DSP-based AR projects discussed in section 4.3.1, one [Naimark and Foxlin 2002] was not concerned with providing graphics, while the other [Uenohara and Kanade 1995] generated only very rudimentary wireframe models.

Similar concerns apply to the two FPGA-based projects [Smith et al 2005, Guimarães et al 2007]. In each case, the focus was on the efficient implementation of a computer vision algorithm, with graphics being a secondary consideration.

[11] http://www.analog.com/processors/blackfin/index.html
[12] http://blackfin.uclinux.org/
[13] http://blackfin.uclinux.org/gf/project/toolchain/
[14] http://www.surveyor.com/blackfin/

4.4.2 Mobile AR plaftorm

While suitable for complex vision algorithms, it is not certain that a solely DSP or FPGA-based platform could handle the necessary graphical duties required of an AR application. Adding a special-purpose 3D graphics accelerator would resolve this issue; however, discrete GPUs are hard to come by.

Fortunately, a solution exists in the form of the applications processor SoCs described in Section 4.2.2. Devices such as the OMAP3430 combine a DSP and a GPU in a single chip, along with a general-purpose ARM CPU. The studies outlined in Section 4.3.2 have already demonstrated AR applications running on this class of devices. In particular, Schmalstieg and Wagner [2007] showed that the OMAP2420 (a precursor to the OMAP3430, with a GPU but no DSP) was capable of rendering a detailed 3D model while tracking markers in the video stream.

As well as its selection of processor cores, there is another reason to favour the OMAP3430 processor. Two inexpensive single-board computers, the Gumstix Overo[15] and the Beagle Board[16], are based on this chip[17]. The benefit of these system is that they provides a generic processor board that can be extended with application-specific modules. This simplifies the hardware development process by making it unnecessary to design the system from scratch. In addition, both the Beagle Board and the Gumstix Overo run Linux and provide an open framework for software development.

The Gumstix and Beagle Board systems share a number of characteristics in addition to their identical processors. They both provide a main board containing the processor, RAM, Flash memory, and power management circuitry. The main difference between the two systems is that the Beagle Board includes a large set of peripherals on the main board, such as USB ports and a Digital Video Interface, while the Gumstix does without, instead making these features available through expansion boards. This allows the Gumstix main board to measure only 17 mm by 58 mm. As Figure 4.3 shows, this is noticeably smaller than the Beagle Board.

4.5 SUMMARY

There is a limited body of research available on embedded AR systems. On the other hand, there is a large amount of published work describing the design and implementation of embedded computer vision platforms. Given the central role computer vision plays in vision-based AR, it is useful to examine such systems.

There are a variety of types of processing elements available to the embedded systems developer. Special-purpose processors such as DSPs can provide good per-

[15] http://www.gumstix.com/
[16] http://beagleboard.org/
[17] Or, rather, the functionally-identical OMAP3530; see Section 5.2.

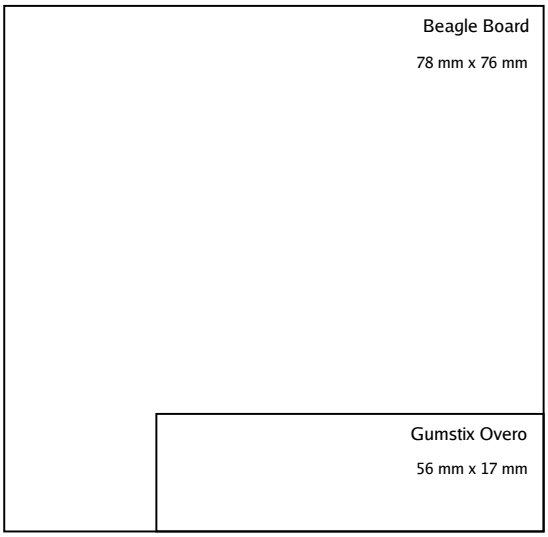


Figure 4.3: Scale diagram of the Gumstix Overo and Beagle Board single-board computers.

formance for specific tasks, while general-purpose processors offer greater flexibility. An embedded system can be built around a single processor or can combine multiple processing types for improved performance.

Section 4.4 presented two designs for an embedded AR platform. The first made use of a Blackfin DSP/microcontroller hybrid processor. The second used the OMAP3430 multi-processor system-on-chip, which provides a DSP and GPU along with a general-purpose ARM processor.

Due to the availability of generic hardware platforms for the OMAP3430 and doubts about the Blackfin's graphics capabilities, a system based on the OMAP3430 processor is the preferred solution. As compactness and light weight are important design factors, the Gumstix Overo single-board computer is better suited to this project than the Beagle Board. The design and implementation of such a system is described in detail in Chapter 5.

Chapter 5

PROTOTYPE DEVELOPMENT

5.1 INTRODUCTION

This chapter presents the development of a prototype embedded augmented reality viewer based on Gumstix Overo single-board computer. The Gumstix Overo computing platform was selected to form the basis of this device for the reasons discussed in Chapter 4, notably its small size and its inclusion of a DSP and graphics accelerator. The Gumstix system and the OMAP3 processor that it uses are described in Section 5.2.

The Gumstix Overo mainboard is paired with a Summit daughterboard. This daughterboard provides a high-definition video output, one of the two capabilities required for the device to serve as a platform for vision-based augmented reality applications. The other necessary capability is a video input, presumably from an attached camera, that provides the live image stream that is to be augmented. As there are no official camera peripherals available for the Gumstix Overo system, a custom camera board has been developed. The OMAP3 processor used by the Gumstix system supports these features through the inclusion of on-board video capture and display hardware, as discussed in Section 5.3.

Section 5.4 describes the development of the custom camera peripheral. The first task is to create the physical circuit board with the necessary components, in this case a camera module along with interfacing circuitry. Following this, software drivers are developed for the Linux kernel, in order to allow the system to access the image data captured by the camera.

The resulting prototype consists of a Gumstix Overo mainboard with a Summit daughterboard and the custom camera board. A monitor is connected to the daughterboard to act as a display and the board is plugged into a wall power supply. The prototype is thus neither completely handheld nor free of cables. Nevertheless, it is sufficient to demonstrate the processing capabilities of the Gumstix Overo system and to evaluate its suitability as a platform for augmented reality applications.

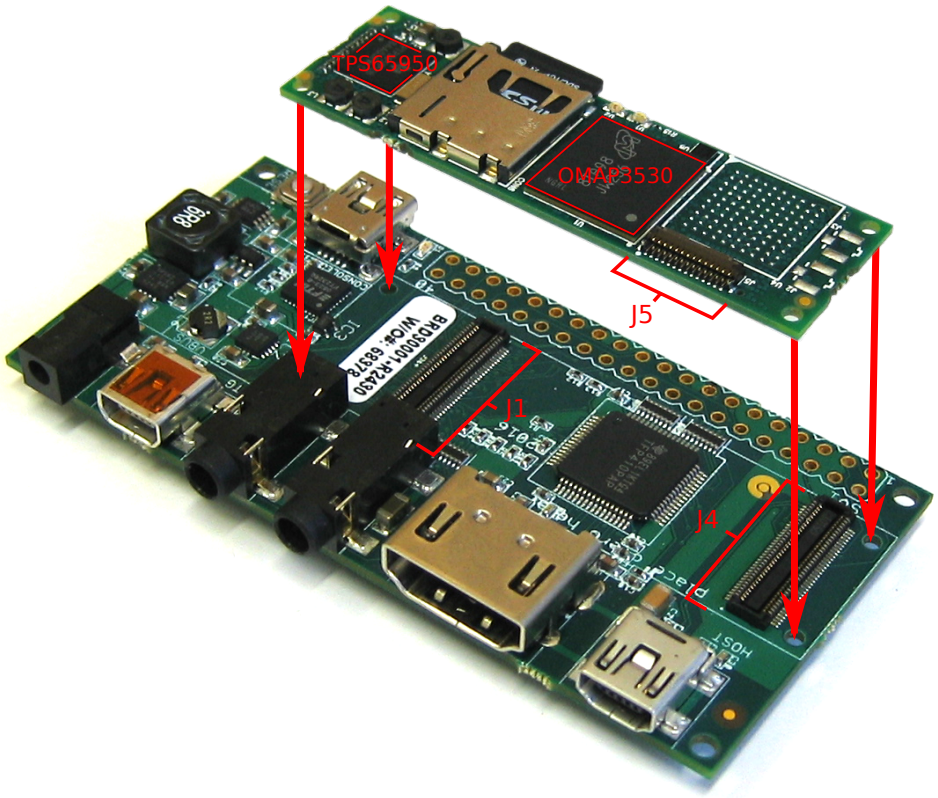


Figure 5.1: The Gumstix Overo Water COM (top) along with the Summit daughterboard. The COM attaches to the daughterboard by connectors J1 and J4; the sockets visible in the image mate with corresponding plugs on the underside of the COM. Connector J5 exposes the camera control signals. The OMAP3530 applications processor is mounted on the COM, with the memory chip stacked on top of it. Also visible on the COM is the MMC card socket and TPS65950 power management IC. The daughterboard provides audio and video output, USB interfaces, and a power socket.

5.2 THE GUMSTIX OVERO COMPUTING PLATFORM

The Gumstix[1] range consists of a variety of ARM-based motherboards and special-purpose daughterboards; using these parts, a developer can quickly assemble a customised system. Each motherboard, or, to use the Gumstix terminology, Computer-On-Module (COM), contains a minimal set of features: a low-power microprocessor, RAM and Flash, an MMC slot, and, as an option, a wireless communications interface. This is coupled with a daughterboard containing additional ports (USB, audio, etc.) or other required capabilities, such as an LCD. A number of variant motherboards are available; along with the many different daughterboards on offer, this provides a large

[1] http://www.gumstix.com/

variety of configurations to choose from. Figure 5.1 shows the configuration employed in this project: an Overo Water COM with a Summit daughterboard.

An appealing feature of the Gumstix system is the effort that its creators have made to offer a relatively "open" platform. The preinstalled operating system is Linux-based, with all source code (including Gumstix-specific modifications) available from a public repository[2]. Comprehensive documentation is provided on the developer website, including user guides and hardware information along with a user-edited wiki. Complementing these resources is the community mailing list[3].

The latest iteration of the Gumstix range, available since October 2008, is the Overo series. These are based on TI's OMAP3 applications processors, with a choice of either the OMAP3530 or the OMAP3503. Both employ an ARM Cortex-A8 core; the OMAP3530 additionally incorporates a DSP and graphics accelerator. In each case, the processor is clocked at 600 MHz and is coupled with 256 MB of RAM and 256 MB of Flash ROM in a Package-on-Package (PoP) arrangement. An MMC slot is included for additional storage and an optional wireless chipset provides Bluetooth and Wi-Fi communications.

On-board expansion connectors provide a method of expanding the functionality of the device with special-purpose expansion boards. The Gumstix Overo COMs each offer three expansion connectors for this purpose, shown in Figure 5.1 labelled J1, J4, and J5. These expose the various functions of the processor:

- J1 (70-pin): LCD, PWM, and analogue signals.
- J4 (70-pin): Extended Memory Bus and MMC signals.
- J5 (27-pin): Camera control signals.

Figure 5.2 shows the specific peripheral features made available by each connector. The 70-pin connectors J1 and J4 are used to attach the COM to a daughterboard, either one of the official ones available from the Gumstix store or a custom design. There are currently no official expansion boards that make use of connector J5.

Power is supplied to the COM via J1 and J4. Daughterboards typically include a jack to allow a power source (3.3 V to 4.2 V) to be plugged in. The 1.8 V required by the processor is produced from the unregulated supply by a TPS65950 integrated power management IC located on the COM. In addition, the TPS65950 provides an audio codec and other miscellaneous features, some of which are exposed on connector J1 (as shown in Figure 5.2).

[2]The Gumstix Overo build system is explained in Section 5.2.2.
[3]https://lists.sourceforge.net/lists/listinfo/gumstix-users

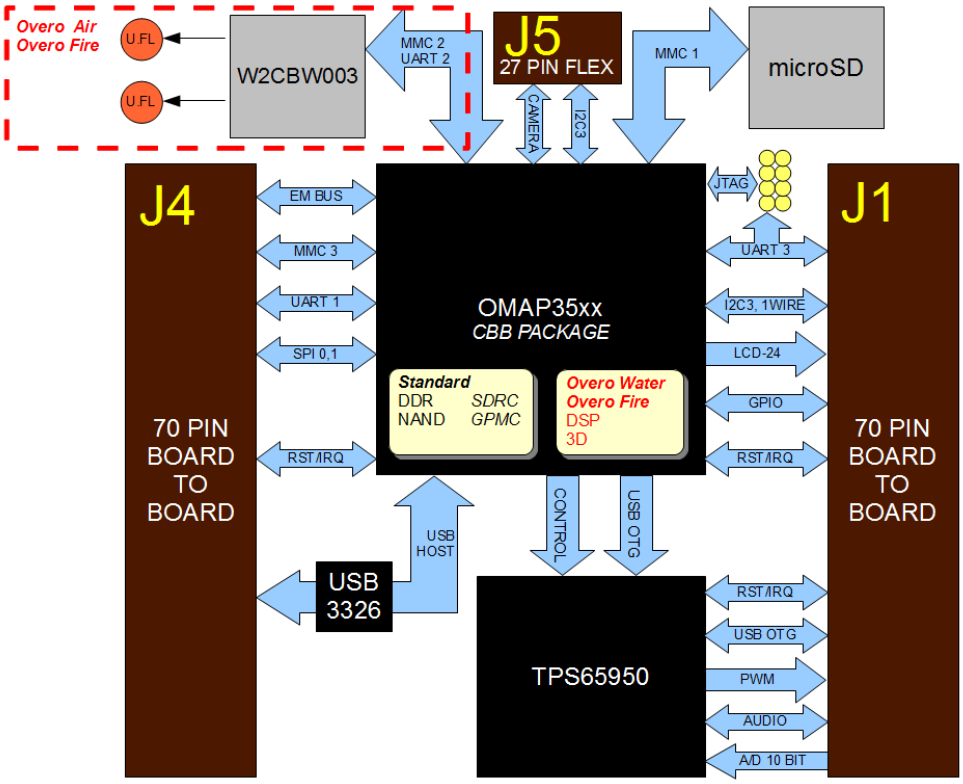


Figure 5.2: Block diagram of Gumstix Overo COMs, taken from the Gumstix Overo COM Signals document V1.2.

5.2.1 The OMAP3530 multimedia applications processor

The Texas Instruments OMAP 3 "applications processors" are a range of integrated systems-on-chips designed for multimedia-oriented devices such as mobile phones. TI makes the OMAP35xx processors available to catalogue distributors, while equivalent models in the OMAP34xx and OMAP36xx designations are supplied directly to handset manufacturers. As such, the Gumstix Overo (Earth and Water[4]) COMs use the OMAP3530, while the Nokia N900, for example, uses the functionally-equivalent OMAP3430. These processors sport a wealth of peripherals and subsystems; the technical reference manual for the OMAP3530[5], for example, is over three and a half thousand pages in length.

The OMAP3530 chip is the most powerful of the OMAP35xx series, including both a TMS320C64x+ DSP and a PowerVR SGX530 graphics accelerator alongside the general-purpose ARM Cortex-A8 CPU. A block diagram of the functional units comprising the OMAP3530 is shown in Figure 5.3. Of particular relevance to this project are the integrated special-purpose processors and the camera interface subsystem, covered in detail in Section 5.3. The on-board DSP and graphics accelerator offer the potential to speed up key parts of the AR processing pipeline. The built-in camera interface subsystem allows an image sensor to be connected directly to the processor, avoiding the overhead of a generic protocol such as USB.

In its 3430 form, this processor powers a number of high-end mobile phones including the Nokia N900, Motorola Droid, and Palm Pre. In addition to its use in consumer devices, there are a number of development and prototyping kits available. Two of these, the Gumstix Overo system and the Beagle Board, were selected for comparison in Section 4.4.2. These boards were both created with a strong community focus and offer well-documented hardware interfaces, freely-available schematics (for the Beagle Board and for the Gumstix Overo expansion boards), and an open build system through OpenEmbedded support.

One consequence of its status as an industry-standard processor is the amount of work that has gone into making sure it is well supported by the Linux kernel. Development was originally carried out in-house by Texas Instruments but has since migrated to an official branch of the mainline kernel[6]. Architecture-specific code for the OMAP3 processors can be found in the arch/arm/mach-omap2 subdirectory of the kernel source tree[7]. Nearly all features of the OMAP3530 processor are supported in the kernel; the one major exception is the camera interface subsystem (discussed

[4]The Fire and Air models use the OMAP3503 instead, which lacks the OMAP3530's DSP and graphics accelerator.

[5]http://www.ti.com/litv/pdf/spruf98g

[6]http://git.kernel.org/?p=linux/kernel/git/tmlind/linux-omap-2.6.git

[7]Why not arch/arm/mach-omap3? The developers felt that the OMAP2 and OMAP3 architectures had more similarities than differences (in contrast to OMAP1 and OMAP2), hence they should share a directory.

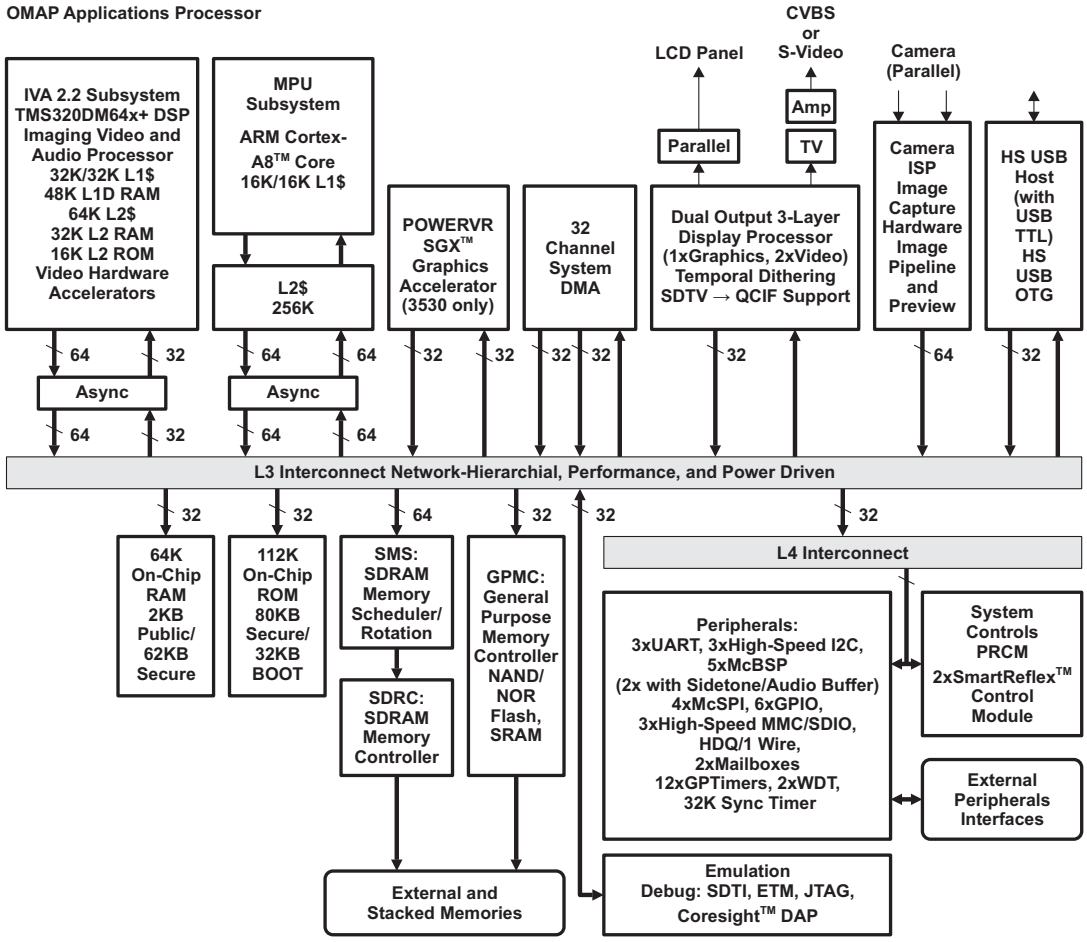


Figure 5.3: Functional block diagram of the OMAP3530 Applications Processor, taken from the OMAP3530 datasheet.

further in Section 5.3.2).

5.2.2 The OpenEmbedded build system

Software development for the Gumstix Overo computers is supported with a build system based on the OpenEmbbedded framework[8]. OpenEmbedded offers a way to automate the process of cross-compiling programs and libraries, or even an entire system image, for embedded Linux-based devices. It plays a similar role to that of a package manager on a traditional desktop operating system, with the distinction that every package is compiled on-demand; thus it resembles the FreeBSD ports collection[9] and Gentoo's "Portage[10]."

Some typical operations that OpenEmbedded is used for are:

- Building a cross-compilation toolchain.
- Compiling a specific program for the target system.
- Compiling a new kernel for the target system.
- Creating a complete system image.

An important feature of OpenEmbedded is its support for cross-compilation, as the target system will typically have a different architecture than the one on which the build system is running. The developer sets configuration variables to define the properties of the target (such as its architecture and preferred kernel version) and OpenEmbedded automatically ensures that all packages are configured and built correctly.

The OpenEmbedded system consists of two main parts: a collection of *recipes*, or scripts that define how to configure and build each package, and the Bitbake tool[11] for interpreting these scripts and executing the relevant tasks. Using Bitbake, a developer can cause any program or library to be built according to the instructions in the corresponding recipe. Bitbake breaks the operation down into a number of simple steps, e.g., 'fetch source,' 'extract source,' 'configure,' 'compile,' etc. These tasks may depend on others (for example, before a program can be compiled the cross-compilation toolchain must be set up). Bitbake will automatically and recursively check that any prerequisites are satisfied and, if not, will cause them to be built before the main task is executed.

[8] http://www.openembedded.org/
[9] http://www.freebsd.org/ports/
[10] http://www.gentoo.org/doc/en/handbook/handbook-x86.xml?part=2&chap=1
[11] http://bitbake.berlios.de/

A branch of OpenEmbedded containing Gumstix-Overo-specific modifications is hosted in a public repository[12]. Instructions for setting up the build system can be found on the Gumstix developers website[13].

5.3 CAPABILITIES OF THE GUMSTIX PLATFORM

The combination of a Gumstix Overo motherboard and a Summit daughterboard forms a capable embedded system suitable for multimedia applications development. The OMAP3 applications processor powering this system provides a general-purpose CPU along with peripherals for handling image capture, graphics rendering, and video output, all of which are employed in augmented reality. While the video output and 3D rendering capabilities are fully supported by the existing Gumstix hardware and software, enabling image capture will require the development of a custom camera module, along with associated drivers.

5.3.1 3D rendering and video output

The Summit daughterboard provides high-definition video output in the form of a DVI (Digital Video Interface) connection, allowing an external monitor to be used as a display. The OMAP3530 includes a display interface subsystem to handle video stream generation. Graphics operations (both 2D and 3D) may be carried out by the general-purpose ARM processor, but, if performance is a concern, a better option is to make use of the on-chip PowerVR SGX graphics processing unit[14].

The display subsystem is fully supported in the Linux kernel. An X11 driver, xf86-video-omapfb[15], exists to allow an X server to run on this device. OpenEmbedded provides recipes for the X.org window server along with various graphical applications.

Support for the PowerVR SGX is not included in the mainline kernel; instead, the OMAP35x graphics SDK supplies the necessary drivers. This SDK is provided by Texas Instruments and contains proprietary kernel modules and graphics libraries, along with documentation and examples. The libraries support the following APIs:

- OpenGL ES 1.1/2.0 (embedded 3D graphics)[16]
- OpenVG 1.1 (vector graphics)[17]
- EGL (native platform graphics interface—X11 and framebuffer backends supported)[18]

[12] http://gitorious.org/gumstix-oe/
[13] http://www.gumstix.net/
[14] http://www.imgtec.com/powervr/sgx.asp
[15] http://cgit.pingu.fi/xf86-video-omapfb/
[16] http://www.khronos.org/opengles/
[17] http://www.khronos.org/openvg/
[18] http://www.khronos.org/egl/

The kernel modules and OpenGL ES graphics library can be built and installed with OpenEmbedded using the omap3-sgx-modules and libgles-omap3 recipes, respectively. However, the OMAP35x graphics SDK must be manually downloaded first, as it requires a registered account on the Texas Instruments website.

5.3.2 Video capture

On all Gumstix Overo COMs there is a connector provided specifically to allow a camera to interface with the processor. This connector exposes various data and control signals (managed by the OMAP3 camera interface subsystem) as well as a single I^2C channel. Although none of the official expansion boards make use of this connection, it is fully documented, allowing for the creation of a custom camera board.

The OMAP3 camera interface subsystem (also referred to as the camera ISP or image signal processor) handles communications with external image sensors. In the simplest scenario, its job is to generate the necessary control signals and transfer the resulting image data to memory. It can also be configured to perform additional operations, including data format conversion and image rescaling.

A high degree of configurability allows the camera ISP to support a variety of different camera modules. It can accept data in raw RGB (i.e., Bayer), YUV, or JPEG-encoded formats. Data can be transferred in parallel either with separate synchronisation signals or with synchronisation codes embedded in the datastream (ITU-R BT.656 protocol)[19]. Finally, multiple bit-depths are supported, with up to 12 bits able to be transferred in parallel.

Three main modules make up the camera ISP:

- The CCDC receives the image data from the sensor and either writes it directly to memory or passes it on to one of the other modules for further processing. The received data may be formatted as raw RGB, YUV 4:2:2, or JPEG but the CCDC does no format conversion itself.
- The previewer can carry out a variety of operations including white balance adjustment and colour correction. It operates on raw RGB image data either drawn from memory or received from the CCDC and produces a result in YUV 4:2:2 format.
- The resizer can up- or down-sample an image by a factor of four or less. It operates on YUV 4:2:2 data, from memory, the CCDC, or the previewer.

Figure 5.4 shows the valid image data paths between these three modules and memory for the various data formats. The format in which it is received by the CCDC

[19] Inspection of the rx51 camera drivers (see Section 5.5.1) reveals a capacity for serial data transfer, though this mode is not documented in the OMAP35xx technical reference manual.

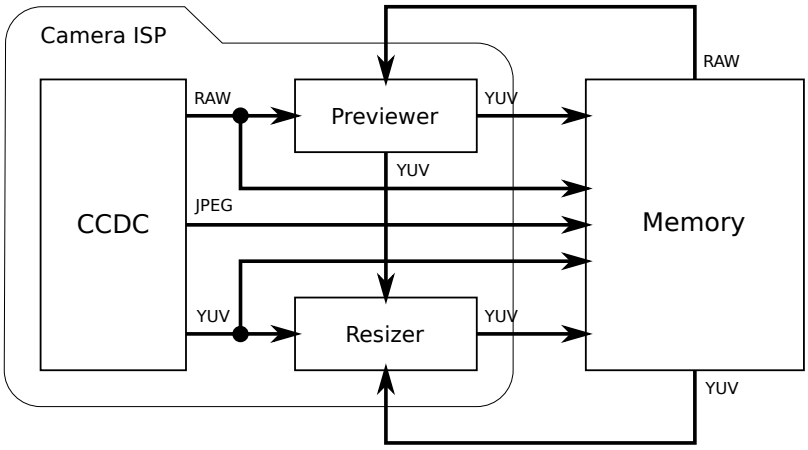


Figure 5.4: An illustration of the possible data paths for image data within the camera ISP and between the ISP and main memory.

determines the possible data paths; for example, YUV data may be written directly to main memory or redirected through the resizer module but cannot be operated on by the previewer (as shown in the diagram).

Two additional modules are available that can collect statistics on the incoming image data, but only when it is in raw RGB format. These are the H3A (hardware auto-exposure/auto-focus/auto-white-balance) and HIST (histogram) units. Raw RGB appears to be the preferred format as it is the only one for which all modules (previewer, resizer and statistics-collection) are available.

Unfortunately, support for the camera interface subsystem is missing from the mainline Linux kernel. However, drivers are being worked on, with the main development effort building on the Nokia N900 camera drivers. This code was originally produced in-house at Nokia to enable the two cameras on their N900 mobile phone. As discussed in Section 5.5, it is now hosted in a publicly-accessible repository and is under active development. The intention is that it will eventually be submitted to the mainline kernel as a generic framework for managing the OMAP3 camera interface subsystem.

5.4 DEVELOPMENT OF THE OVEROCAM CAMERA BOARD

Expansion boards for the Gumstix Overo COM provide a variety of extra features but, at the time of writing (September 2010), the official range does not include a camera board[20]. This made it necessary to develop a custom camera board designed to connect to the camera interface on the Gumstix Overo COMs, along with drivers for the image

[20]There is, however, a third-party camera board being offered by e-con Systems (http://www.e-consystems.com/omapovero.asp). This unit had not been released at the time the decision was made to create a custom camera board, otherwise it would have been used instead.

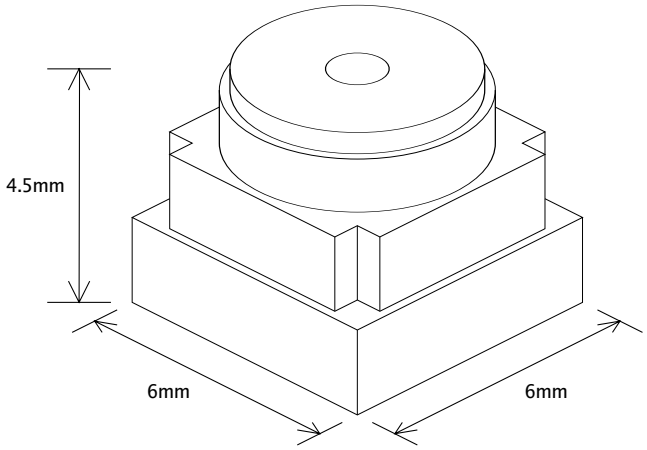


Figure 5.5: The TCM8230MD VGA camera module is a compact all-in-one unit with a footprint of 6 mm square and a height of 4.5 mm including integrated lens assembly.

sensor and OMAP3 camera interface subsystem. This board has been given the name *Overocam*.

5.4.1 The camera module

The requirements of the camera board were simple: to supply a colour video stream at a resolution of 640×480 and 30 frames per second. These are the parameters of a typical USB webcam, as used for desktop AR applications.

At the time, image sensors were not generally available through standard distributors such as Farnell. A solution was discovered in the form of the TCM8230MD camera module available from SparkFun Electronics[21], an online store specialising in supplying components for hobbyists. It was priced as USD $9.95 per unit, complete with datasheet.

The Toshiba TCM8230MD VGA camera module captures images at the necessary resolution and framerate. Its small size is indicated in Figure 5.5. Colour images are delivered with 16 bits per pixel across an 8-bit parallel bus, in either YUV 4:2:2 or RGB 5:6:5 format. An on-board signal processor offers auto electrical shutter control, auto gain control, and auto white balance.

5.4.2 Interfacing the processor and the camera

The control and data transfer scheme employed by the TCM8230MD follows a standard pattern for image sensors. Image data is sent across the parallel data bus, with the aid of separate frame, line, and pixel synchronisation signals. The task of monitoring these synchronisation signals and receiving the data from the bus is handled by the OMAP3 camera ISP. Specific settings, such as image resolution and data format, are

[21] http://www.sparkfun.com/

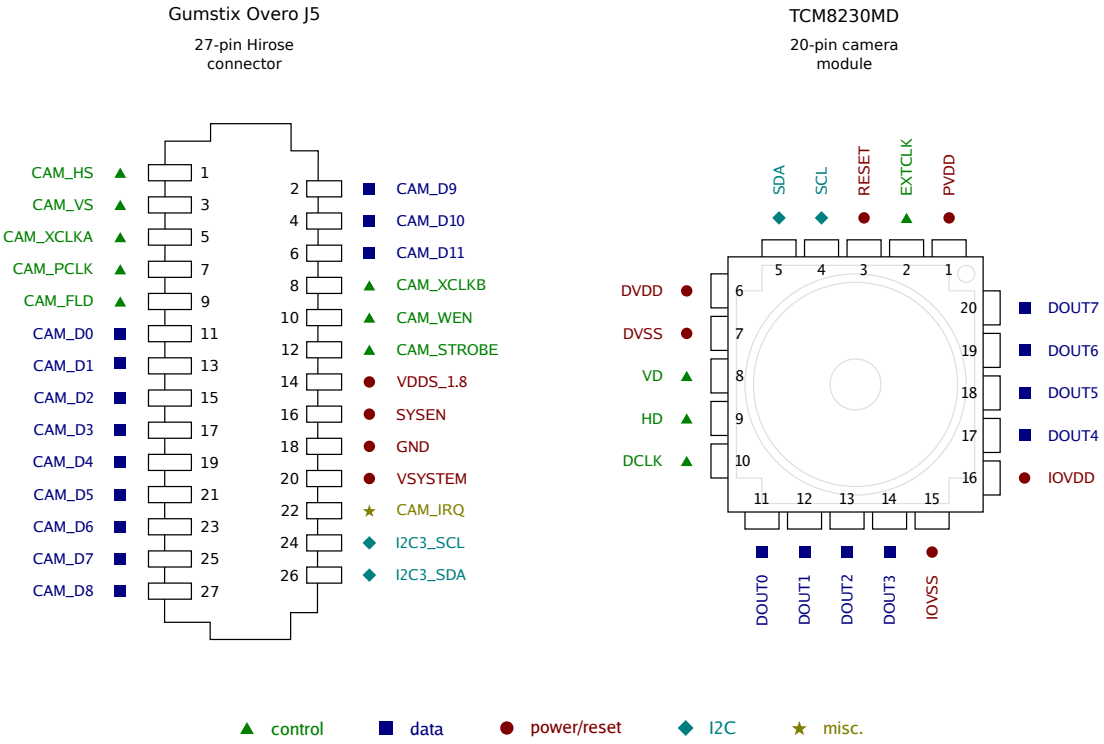


Figure 5.6: Pins belonging to the Gumstix Overo camera interface connector J5 (left) and the TCM8230MD camera module (right). Note that CAM_IRQ is not included among the control signals as it is, in fact, a regular GPIO pin (the "CAM_IRQ" label was invented by the Gumstix developers and does not appear in the OMAP35xx datasheet).

set by manipulating the values of internal registers accessible via the I^2C interface. As described in section 5.3.2, the OMAP3 camera ISP manages the control and data signals (with the exception of the I^2C interface). The Gumstix Overo camera connector exposes the pins associated with the camera ISP as well as one of the OMAP3530's four I^2C interfaces, along with power and reset lines. Figure 5.6 shows these pins alongside those of the TCM8230MD, broken down by category (control, data, power/reset, I^2C, and misc.).

The camera board circuit design is conceptually simple. As Table 5.1 shows, the majority of the pins on the image sensor have a one-to-one correspondence with the pins on the Gumstix Overo camera connector. Differences in the voltage levels between the TCM8230MD (2.8V) and the OMAP3530 (1.8V) preclude the direct connection of these pins, making it necessary to install level converters on each line. A separate level converter is required for the I^2C bus and two LDO (low dropout) voltage regulators provide the camera with necessary voltage levels. The complete circuit schematic, PCB layout and parts list are provided in Appendix A.

A high-level view of the Overocam camera board is given in Figure 5.7, showing the circuitry required to interface the TCM8230MD camera module to the Gumstix

Connector J5		TCM8230MD
CAM_HS	←	HD
CAM_VS	←	VD
CAM_FLD		
CAM_PCLK	←	DCLK
CAM_WEN		
CAM_STROBE		
CAM_XCLKA	→	EXTCLK
CAM_XCLKB		
CAM_D[0:7]	←	DOUT[0:7]
CAM_D[8:11]		
I2C3_SCL	⇌	SCL
I2C3_SDA	⇌	SDA

Table 5.1: The control, data, and I2C signals exposed by the Gumstix camera connector and their correspondents on the camera module. A logical (not physical; see text) connection is indicated by an arrow from the output to the input. The I^2C interface uses open-collector outputs, hence the connection is bidirectional.

processor. Conversion of the data and control signals (except the I^2C interface) is performed by two MAX3002EUP 8-channel level translators. These devices are designed to be fully bidirectional, meaning that each channel senses when either of its ports is being driven (high or low) and sets the other accordingly; no explicit direction signal is required. This characteristic reduces the number of chips required, since a single chip can handle signals in both directions. It also make them unsuitable for open-collector signals, necessitating a dedicated converter for the I^2C interface. The PCA9306 I^2C level shifter was employed for this purpose.

The TCM8230MD camera module requires two distinct supply voltages to operate: a high voltage, (either 2.8 V or 2.5 V) for the sensor photodiode and external I/O, and a low voltage (1.5 V) for the sensor A/D converter and internal digital circuitry. Since neither of these levels are compatible with the 1.8 V provided by the Gumstix board, they must both be generated from the unregulated Gumstix power supply (approximately 3.3 V to 4.2 V). This is accomplished by two LDO regulators, one of which produces 2.85 V (well within the ±0.2 V tolerance of the camera module) and the other 1.5 V. These regulators have an output voltage accuracy of 2% and a dropout voltage of 160 mV.

Figure 5.8 shows the assembled "Overocam" camera board. Note the camera connector, which attaches to J5 (see Figure 5.1) by a short ribbon cable (not pictured). Much of the area of the board is taken up by the tracks for the data and control signals. This could be reduced by using a greater number of layers; the current design uses two layers only.

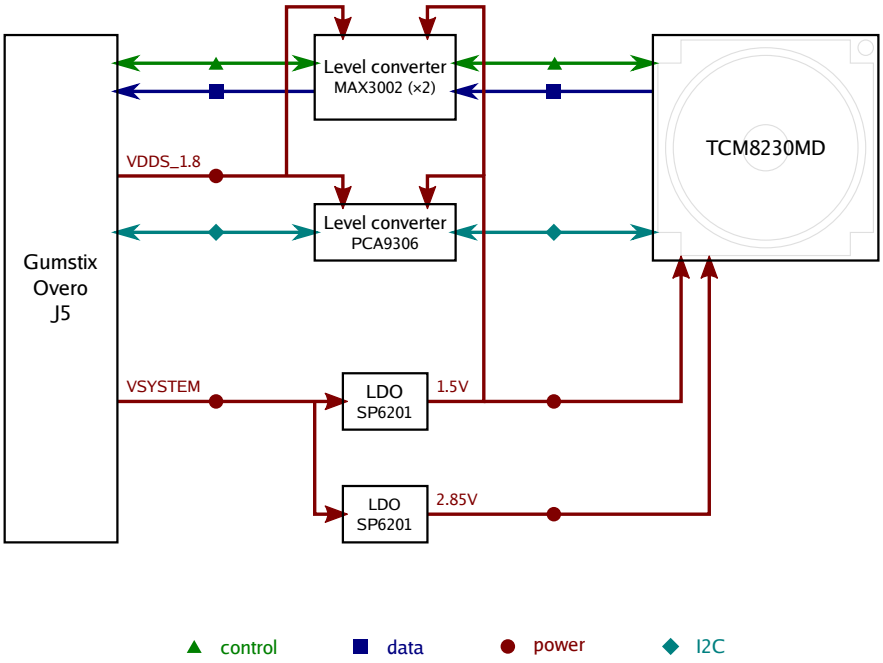


Figure 5.7: High-level view of the Overocam camera board. Arrows indicate logic and power connections. Reset/enable signals are not shown.

Figure 5.8: The "Overocam" camera board. Visible in this photo are the camera interface connector (lower centre), camera chip (upper right), logic level translators (the two large ICs), and LDO voltage regulators (the two smaller ICs); the I^2C converter is located on the underside of the board.

5.5 OVEROCAM SOFTWARE SUPPORT

The OMAP3 processor enjoys good overall Linux support, with, unfortunately, the exception of the camera ISP. At the time of writing, drivers for the camera interface subsystem are absent from the mainline kernel. Nokia has spearheaded an effort to produce the necessary drivers to make the two cameras on their N900 smartphone device usable. These drivers are in an experimental state and considerable work is necessary if they are to be used for any hardware other than the N900. This section describes the structure of these drivers, along with the modifications that were required to make them work with the Overocam camera board.

5.5.1 Experimental camera drivers for the RX51

Nokia's Linux-based N900 (called the RX51 during development) smartphone device employs the OMAP3430[22] as its processor and includes two cameras among its peripherals. The main one is outwards-facing and is intended for taking photos, the other is pointed at the user, presumably to enable video phone calling. The first (called "Stingray" in the driver code) has a high-resolution (5 megapixel) image sensor, an LED flash, and an automated lens. The second ("AcmeLite") provides only VGA resolution images, does not have a flash, and has a fixed lens.

Development of kernel support for the OMAP3 camera ISP and the N900 camera hardware is carried out in a publicly-accessible repository. Until 27 August 2010 this repository was hosted at http://gitorious.org/omap3camera; the new location is http://meego.gitorious.org/maemo-multimedia/omap3isp-rx51. The discussion in this section refers to the older omap3camera version as of May 2010 and does not reflect more recent changes.

The omap3camera drivers conform to the Video4Linux version two (V4L2) specification[23]. This document specifies a common infrastructure for handling video and radio devices within the Linux kernel. It provides a framework for describing any particular configuration of hardware that functions as a video or radio source or sink, along with generic operations for ascertaining its capabilities and controlling its behaviour.

V4L2 drivers share a common structure of having a single main device instance coupled with a number of sub-devices representing supporting components. In the case of the omap3camera drivers, the main device is the OMAP3 camera ISP, represented by the omap34xxcam module. Four sub-devices are present: the et8ek8 and smia-sensor image sensor drivers, the ad5820 lens actuator driver, and the adp1653 LED flash driver.

The Stingray camera accounts for three of the four sub-devices, corresponding to the ET8EK8 image sensor, the AD5820 lens driver chip, and the ADP1653 flash driver

[22] The OMAP3430 is functionally equivalent to the OMAP3530, as explained in Section 5.2.1.
[23] http://v4l2spec.bytesex.org/spec/

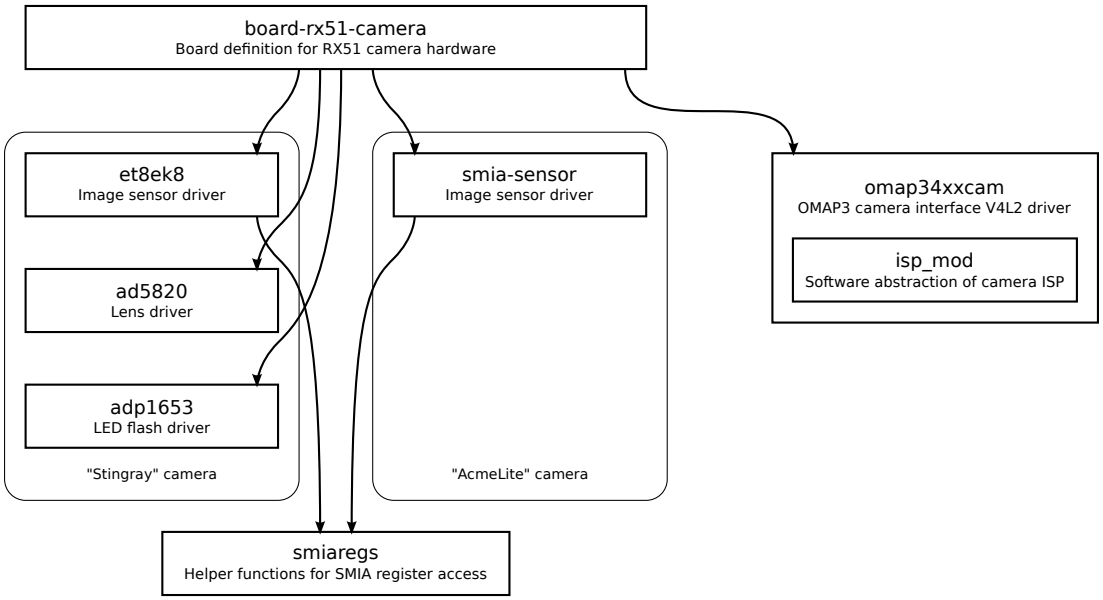


Figure 5.9: The kernel modules provided by omap3camera to support the Nokia RX51 cameras. Arrows in the diagram show the main dependency relationships.

chip. The remaining sub-device is the generic smia-sensor[24], used by the AcmeLite camera. Both et8ek8 and smia-sensor depend on the helper functions defined in smiaregs for operation, though smiaregs is not itself a V4L2 sub-device.

The omap34xxcam driver is, in principle at least, a device-independent module that exposes the camera interface as a V4L2 video source. It is implemented as a wrapper for the isp_mod module, which is a software abstraction of the camera ISP hardware and offers an API for configuring and inspecting this subsystem. The driver does not include specific references to any particular sub-devices; instead, the device configuration is defined in the board-rx51-camera module. However, it does include the hard-coded assumption that it will be managing two cameras, each of which will consist of up to three sub-devices.

5.5.2 Development of custom drivers

Figure 5.9 shows the complete set of modules provided by omap3camera. These modules depend on each other as indicated by the arrows. Figure 5.10 shows an equivalent diagram of the modules required for the Overocam board. The RX51-specific modules (including the board definition and image sensor drivers) have been replaced, while the OMAP3 V4L2 driver has been retained. Unfortunately, this V4L2 driver includes a number of hard-coded values and routines that are specific to the RX51 setup, hence it must be adapted for use with the Overocam board.

[24]SMIA stands for Standard Mobile Imaging Architecture, released in 2004 by Nokia and STMicroelectronics.

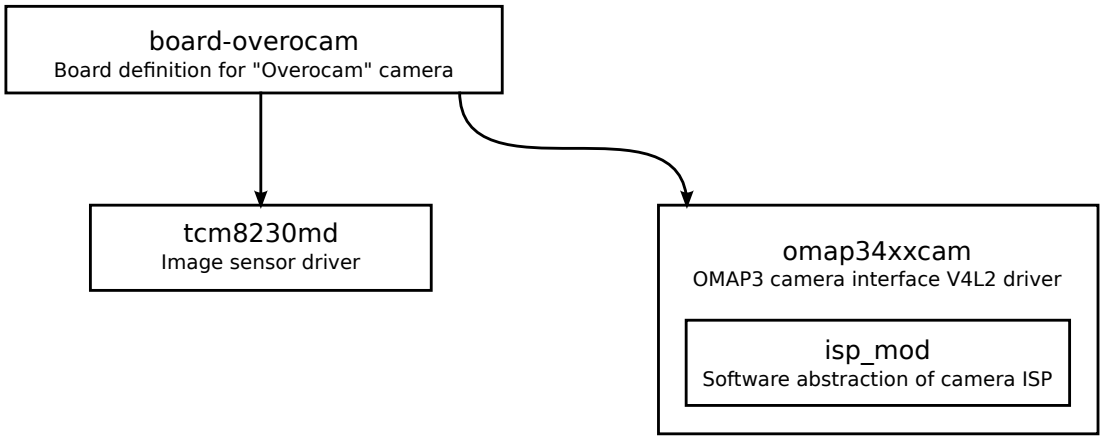


Figure 5.10: The kernel modules that support the Overocam board (c.f. Figure 5.9).

The changes that need to be made to the omap3camera can be broken down into three tasks:

- Create the board-overocam board definition module, replacing board-rx51-camera.
- Create the tcm8230md camera module driver, replacing the RX51-specific chip drivers.
- Modify the omap34xxcam driver and ISP code to work with the TCM8230MD.

The board definition module board-overocam replaces board-rx51-camera. This module is simpler than board-rx51-camera due to the fact that the Overocam board has only a single permanently-enabled camera. This is in contrast to the RX51's dual-camera system, each of which is activated and deactivated independently of the other. Lacking separate lens and flash controllers, the Overocam camera instantiates a single sub-device (for the TCM8230MD camera module) only.

Configuration parameters for the camera ISP driver are specified in the board definition module. These configuration parameters specify the transfer mode used to move data between the image sensor and the camera ISP. The Overocam case uses the parallel data transfer mode instead of one of the more complex serial modes required by the RX51.

The tcm8230md module is a minimal driver for the TCM8230MD camera chip. It supports only the YUV 4:2:2 colour format[25] at 640×480 resolution and 30 frames per second. It does not expose any of the chip's special features (e.g., auto electrical shutter control, auto gain control, auto white balance).

[25]The TCM8230MD can also provide data in RGB 5:6:5 (i.e., non-raw) format, but this is not recognised by the camera ISP.

The TCM8230MD driver follows the structure of the et8ek8 module upon which it is based. It largely consists of definitions of V4L2 operations that allow it to function as a V4L2 device. The V4L2 operations implemented in the tcm8230md module are:

- Set configuration.
- Enable/disable power.
- Get/try/set format (YUV 4:2:2 only).
- Enumerate supported formats/frame-sizes/frame-intervals.
- Activate/deactivate video streaming.
- Get/set parameters (though none supported).
- Get/set pad format.

The pad format operations are part of the new media controller framework[26]. This is an API introduced by the omap3camera developers to manage multiple interconnected video (and, potentially, audio) elements in a consistent manner. It introduces the concept of "media pads" and "media entities." Media entities expose one or more pads, representing video sources or sinks. A set of video entities are configured by forming links between their various pads.

In the Overocam and RX51 systems, the image sensor and camera ISP are each a single media entity. Furthermore, the V4L2 device node (representing the userspace interface) is itself a media entity. The image sensor has a single video output pad, while the device node has a single video input pad. The camera ISP has multiple pads corresponding to its various video sources and sinks; different combinations of links between these pads enable the different datapaths that were shown in Figure 5.4.

At present, two links are formed: from the image sensor to the camera ISP CCDC input and from the CCDC output to the video device node. This is the simplest pathway, as it bypasses the camera ISP previewer and resizer completely. A number of modifications were made to the ISP driver to allow it to receive data in the YUV format, instead of the RGB format used by the RX51 cameras.

Appendix B presents the specific changes that were made to adapt the omap3camera camera drivers to the Overocam board.

5.6 SUMMARY

Chapter 4 introduced the Gumstix Overo platform. Various features of this device suggest it as a suitable platform for augmented reality applications, but the main one

[26]More information on the media controller framework may be found in the thread stemming from the release announcement message archived at http://lwn.net/Articles/417479/.

is the OMAP3 applications processor. This processor has built-in 3D rendering and video capture capabilities that will be of use in this project.

The one component required for augmented reality that is missing from the Gumstix system is a camera peripheral. A custom camera expansion board has been developed to fill this role. A circuit was designed around the TCM8230MD VGA camera module and drivers developed based on those used in the Nokia N900 internet tablet.

Using this camera board, the Gumstix Overo computer can capture images at a resolution of 640×480 and at a framerate of 30 fps. Power tests revealed that, given a 4 V supply, the device consumes 1.28 W when idle and 1.64 W when the camera is running.

Chapter 6

ARTOOLKIT DEMO

6.1 INTRODUCTION

This chapter presents an evaluation of the performance of the completed prototype device. A demonstration AR application was created based on the ARToolKit marker-tracking code. The framerates thus obtained provide an indication of the device's viability as a platform for augmented reality applications.

The AR application looks for a single marker with a predetermined pattern. When it finds this marker, it renders a simple virtual object consisting of a spinning logo on top of the marker. Figure 6.1 shows the augmented scene produced by the application when the marker is found.

The main purpose of the application is to test the marker-tracking performance of the prototype device. Only the single-marker case was tested; multiple markers will increase the workload [Zhang et al 2002], reducing performance. Due to the simplicity of the virtual object, the 3D graphics capabilities of the system are not seriously tested.

The application uses the ARToolKit marker detection and registration algorithms. The marker-tracking method implemented by these algorithms, along with its strengths and weaknesses, was described in Section 3.2.

The application is instrumented with a framerate-reporting facility used to obtain the timing figures reported in this chapter. It employs clock_gettime()[1] for real-time measurement. When run, the application prints out its average framerate once per second.

Where possible, ARToolKit code is reused in the test application. In practice, this is limited to the marker tracking code and the GStreamer video capture back-end. The ARToolKit routines for window management and event handling are not usable due to differences in the graphics environment on the embedded device, as discussed in Section 6.2. It is for this reason that it is not possible to merely recompile one of the existing ARToolKit applications.

[1] http://www.kernel.org/doc/man-pages/online/pages/man2/clock_gettime.2.html

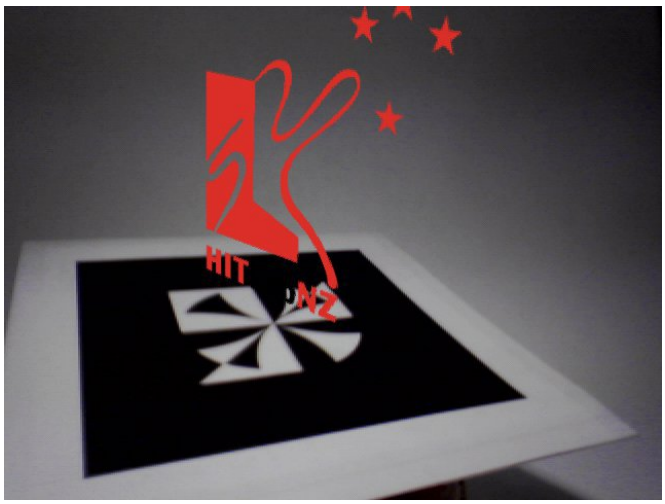


Figure 6.1: A screenshot of the AR test application running on the prototype device.

GStreamer, the multimedia framework employed by ARToolKit to read the video stream from the camera, is available for the device. One reason that this framework is used is that it can be configured to automatically convert the camera's YUV images to the RGB colourspace, as required by OpenGL. The drawback of using GStreamer is that it adds an unknown amount of overhead. An alternative video back-end has been developed that reads image data directly from the Video4Linux2 interface. Section 6.3 presents the results of tests comparing the different video back-ends, as well as different methods for handling colourspace conversion.

In Section 6.4, the various tasks performed by the application are examined to discover their contribution to the overall processing load. The tasks that have the greatest impact on performance should be the focus of future optimisation efforts.

The code for the test application is presented in Appendix C.

6.2 ALTERNATIVE GRAPHICS ROUTINES

In Chapter 3, the demo applications supplied with the ARToolKit library were used to test the performance of the ARToolKit marker-tracking routines. Unfortunately, the same demo applications may not be used to test the prototype device as they depend on libraries that are not available on the embedded system. The solution to this problem is to develop a new test application that uses different libraries.

The missing libraries relate to the 3D graphics stack. As explained in Section 5.3.1, the OMAP3530 processor provides OpenGL ES only, instead of the full OpenGL specification. This is a variant of OpenGL designed specifically for embedded systems, taking into account the limited resources available in such devices. OpenGL ES retains compatibility with the OpenGL wherever possible, offering a subset of the functionality of the full version [Munshi et al 2009]. ARToolKit is written with the expectation

that full OpenGL, along with associated libraries (such as GLUT), is available; as a consequence, parts of ARToolKit will not work on the OMAP3530.

ARToolKit provides a platform-independent abstraction for handling the graphical user interface in the form of the gsub routines. Tasks supported by this abstraction include the following:

- Creating the main application window.
- Registering actions for handling window events (mouse, keyboard and redraw).
- Helper functions for graphics rendering, such as functions for setting the draw mode (2D/3D) and for drawing the video frame background.

These routines make use of the OpenGL Utility Toolkit (GLUT)[2], a cross-platform library providing an abstraction layer for window management and event handling. However, GLUT is not available on the prototype device because of differences between OpenGL ES and full OpenGL. As a consequence, the gsub routines cannot be used and alternative implementations of the above tasks must be developed.

Window creation and event handling is carried out by addressing the windowing system directly instead of using the wrapper functions provided by GLUT. On most Linux-based systems, including the prototype device, this is X11. The Xlib library provides the necessary routines.

One of the tasks performed by GLUT is to communicate with the platform's underlying windowing system to establish an OpenGL rendering context. On regular computers, this involves a call to the platform-specific mechanism (such as GLX for X11 or CGL for Mac OS X). OpenGL ES replaces these facilities with the platform-agnostic EGL library. This library is available for the OMAP3530 and is used by the test application.

The ARToolKit gsub routines include functions for drawing the video frame into the window background and for setting the drawing mode to 2D (i.e., orthographic) or 3D (i.e., perspective) projection. These operations do not rely on GLUT, but they do make use of OpenGL features not provided in OpenGL ES. A typical usage of these functions is as follows:

1. Set drawing mode to 2D (argDrawMode2D()).
2. Draw video frame (argDispImage()).
3. Set drawing mode to 3D (argDrawMode3D()).
4. Draw virtual object.

[2] http://www.opengl.org/resources/libraries/glut/

The final step does not involve the ARToolKit graphics library; instead, the programmer makes the necessary calls to OpenGL to render the virtual object.

The argDrawMode2D() and argDrawMode3D() functions set the OpenGL projection mode differently depending on the type of drawing that is to follow. For 2D drawing, orthographic projection is enabled; for 3D, normal (i.e., perspective) projection is used instead. Traditionally, OpenGL has included a built-in projection matrix that can be changed through calls such as glLoadMatrix(). OpenGL ES has done away with many of the fixed-function parts of the rendering pipeline[3], including the projection matrix, making it necessary to recreate this functionality in the vertex shader.

To draw the video frame background, a rectangular mesh is created and the video image is used to texture it. The argDispImage() function also reverses the lens distortion; for the sake of simplicity, this feature is not present in the test application. ARToolKit does not provide any specific functions for drawing the virtual object; it is up to the programmer to make the necessary OpenGL calls.

Finally, ARToolKit provides functions for converting the camera parameters and marker transformation matrix to OpenGL-compatible projection and modelview matrices respectively. This operation is nontrivial due to the different view model assumed by ARToolKit, however, since it does not rely on any external libraries, the conversion routines can be reused without modification.

6.3 VIDEO-HANDLING VARIANTS

ARToolKit includes a facility for deciding at compile-time which video capture back-end to use. On Linux-based systems there are two options: Video4Linux (the Linux kernel video subsystem) or the GStreamer media framework. The video drivers described in Section 5.5 are Video4Linux drivers; however, they conform to version 2 of this API, whereas the ARToolKit video back-end uses the (obsolete) original version.

GStreamer is a generic multimedia subsystem. It can take inputs from a variety of sources including a file, a TCP socket or a Video4Linux source. Services such as video encoding/decoding, resampling and colourspace conversion are made available through a set of plugin modules. While this system provides a great deal of flexibility, it necessarily adds an uncertain amount of overhead. A simpler alternative would be to receive video data directly via the Video4Linux interface. As the camera driver was written for version two of this interface, an updated Video4Linux back-end has been written.

The set of available image colourspaces is another factor that influences the choice of video back-end. When using the GStreamer back-end the 24-bit RGB format is assumed; any necessary conversion is performed by the GStreamer pipeline before it

[3]The fixed-function pipeline was removed from OpenGL ES in version 2.0; earlier versions retained this functionality. The test application uses OpenGL ES version 2.0.

reaches ARToolKit. The RGB format is convenient for drawing the background as OpenGL only accepts RGB textures, thus no further conversion is necessary.

For the marker detection process, which makes use of image intensity values only, the YUV format would be more efficient. Since the camera's native format is YUV, it is preferable to provide the unmodified image data, delaying conversion to RGB until necessary for drawing. This conversion may be carried out either using a software algorithm or by programming the graphics accelerator.

Five different variants of the test application have been created, three of which use the GStreamer back-end, with the other two using Video4Linux2. For each back-end, there are two variants that use the YUV colourspace: one performs the conversion to RGB in software, the other using the graphics accelerator. The final variant uses the GStreamer back-end with the RGB colourspace, thus the application does not need to perform colour conversion.

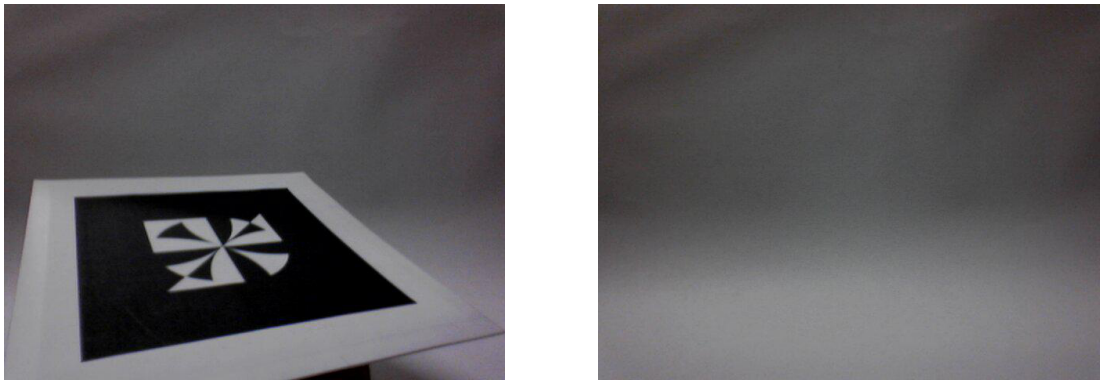
The five variants are:

- gst-rgb: uses the GStreamer back-end with the RGB colourspace.
- gst-swconv: uses the GStreamer back-end with the YUV colourspace, converting to RGB in software.
- gst-hwconv: uses the GStreamer back-end with the YUV colourspace, converting to RGB on the graphics accelerator.
- v4l2-swconv: uses the Video4Linux2 back-end, converting to RGB in software.
- v4l2-hwconv: uses the Video4Linux2 back-end, converting to RGB on the graphics accelerator.

Each variant was tested on two different scenes and the framerate reported by the application was recorded. The two scenes are shown in Figure 6.2; in one a marker is visible, in the other it is not. By keeping the camera stationary and not moving any part of the scene, the view remained constant throughout the tests.

Figure 6.3 shows a graph of the test results. In each case, the scene with the marker took longer to process than the one without. This is explained by the fact that the presence of the marker requires the application to register the marker then render the virtual object, both of which take time. The results also confirm that the Video4Linux2 back-end is faster than GStreamer and that graphics-accelerator-based colour conversion is faster that a software algorithm. When using the GStreamer back-end, the choice of RGB or YUV for the colourspace does not make a significant difference.

Table 6.1 lists the numerical values plotted in Figure 6.3. The V4L2 back-end shows a marked performance improvement over the GStreamer one, reducing the time



(a) (b)

Figure 6.2: Still images from the two scenes used to test the performance of the AR application, both (a) with the marker in view and (b) without.

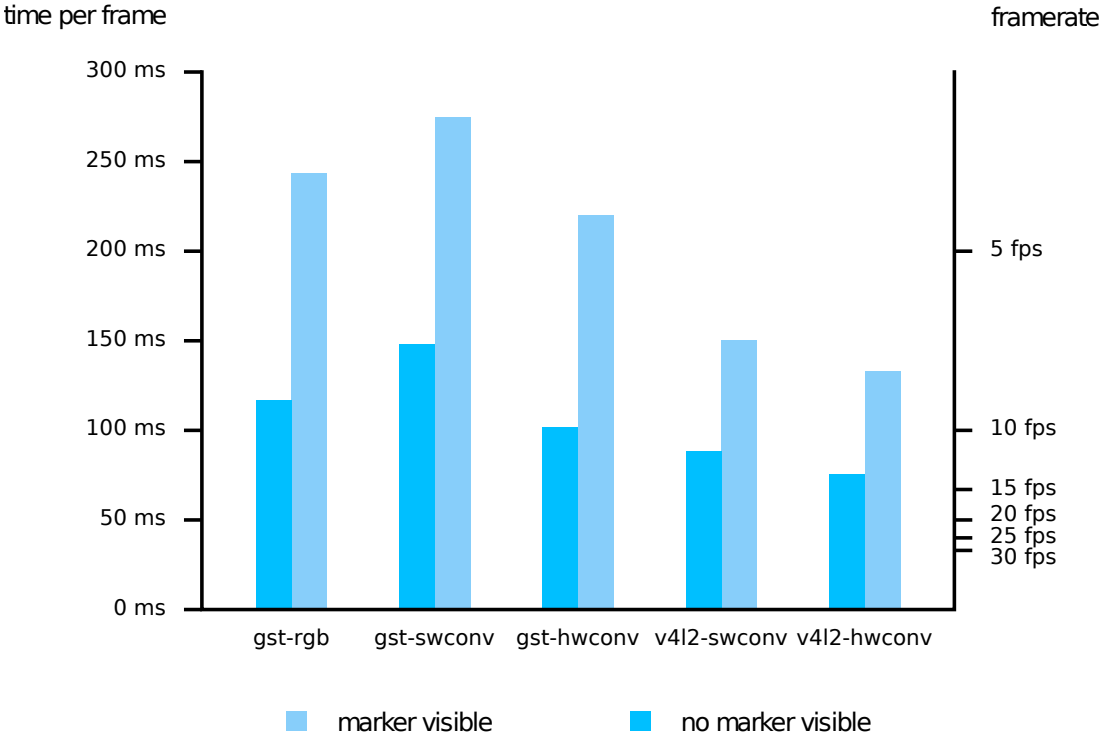


Figure 6.3: Performance measurements for the different variants of the AR application, showing timing information for scenes both with and without a marker present.

	Marker	No marker
gst-rgb	244 ms (4.1 fps)	117 ms (8.5 fps)
gst-swconv	275 ms (3.6 fps)	148 ms (6.7 fps)
gst-hwconv	220 ms (4.5 fps)	102 ms (9.8 fps)
v4l2-swconv	150 ms (6.7 fps)	88 ms (11.3 fps)
v4l2-hwconv	133 ms (7.5 fps)	76 ms (13.2 fps)

Table 6.1: Performance measurements for the different variants of the AR application, in milliseconds per frame and frames per second.

to process a frame by 26 to 125 ms (depending on whether a marker is present and whether hardware conversion is used). Making use of the hardware graphics accelerator for colour conversion saves 12 to 55 ms. Overall, the best results are given by the v4l2-hwconv variant; at 7.5 fps with a visible marker and 13.2 fps without, it is twice as fast as gst-swconv, the slowest variant.

6.4 BREAKDOWN OF PROCESSING STAGES

The AR pipeline implemented in the test application consists of a number of distinct stages, each of which adds to the time required to process a frame. The overall time per frame, hence the application framerate, is determined by the sum of the individual stages. In this section, the impact of each of these stages on the overall performance is measured.

The method used to obtain these measurements is to selectively disable specific tasks and record the change in processing time. Some tasks, such as background rendering, can be disabled without influencing the rest of the program. Other tasks are prerequisites for later stages; for example, virtual object rendering relies on the calculated marker pose. In this case, any dependent tasks must be disabled as well.

Figure 6.4 shows a simplified representation of the major tasks carried out in a single cycle of the test application processing loop. The arrows in the diagram indicate which tasks must be completed before another may begin. Note that, even though the diagram is drawn with two parallel pathways, the application is single-threaded.

Five tests were carried out, with a different selection of tasks enabled in each one. The test scene was the one with the marker visible shown in Figure 6.2. As before, the camera remained stationary and the scene was static. The Video4Linux2 back-end was used, with hardware colourspace conversion.

The results of the tests are presented in Table 6.2. Comparing these measurements to each other yields the following values for the individual tasks:

- Detect markers: 31 ms (23%).
- Calculate marker pose: 40 ms (30%).

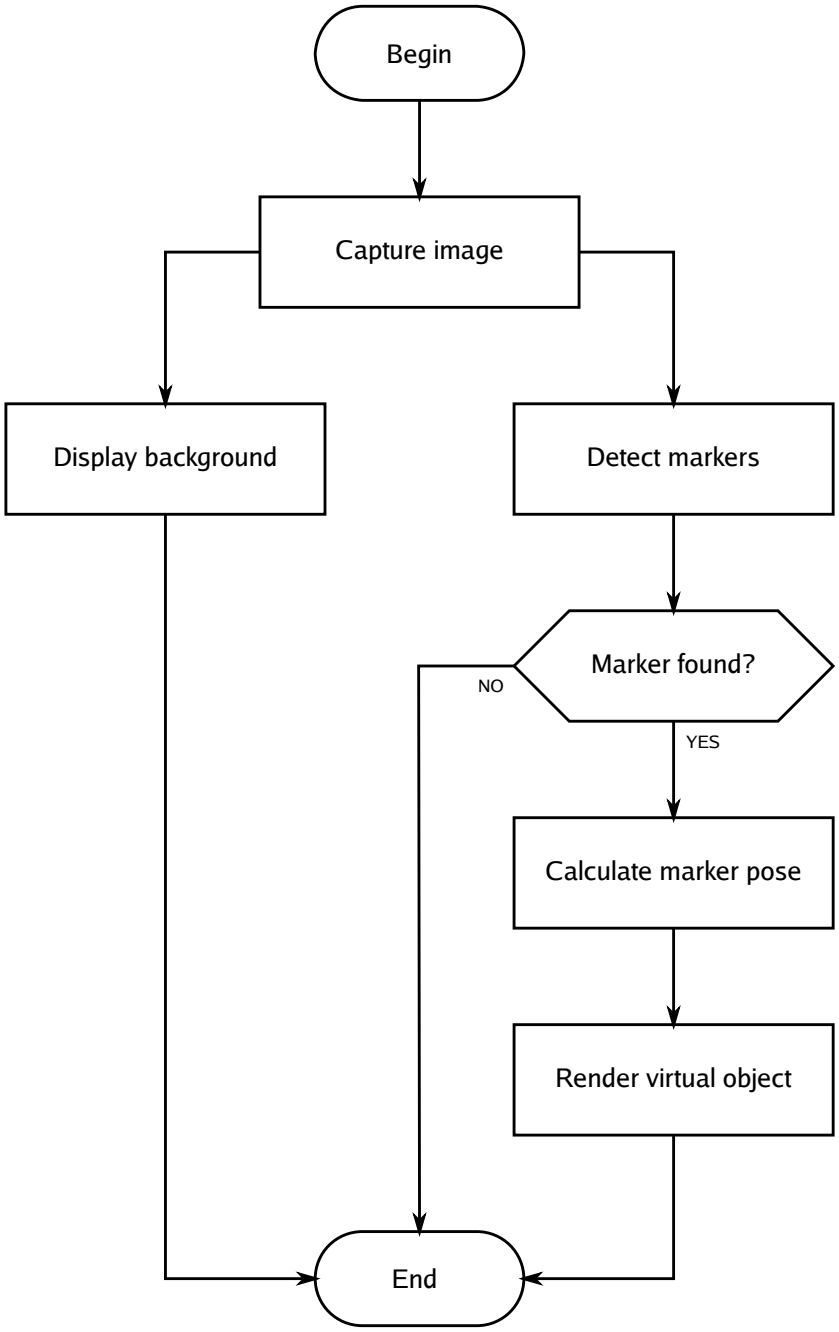


Figure 6.4: A graphical representation of a single cycle of the main processing loop, showing the important processing stages.

	Test 1	Test 2	Test 3	Test 4	Test 5
Detect markers in frame	●	●	●	○	●
Calculate marker pose	●	●	○	○	●
Draw virtual object	●	○	○	○	●
Display video frame background	●	●	●	●	○
Time per frame:	133 ms	124 ms	84 ms	53 ms	95 ms

Table 6.2: Results of tests to determine the cost of the different tasks in the AR processing pipeline (● = task enabled, ○ = task disabled).

- Render virtual object: 9 ms (7%).
- Display background: 38 ms (29%).
- Other tasks: 15 ms (11%).

These results show that, apart from the virtual object rendering, the other three major tasks consume a similar amount of time. Marker pose calculation and backround display take the most time at around 40 ms, or 30% of the total processing time, each. The virtual object used in these tests is simple, however, consisting only of a textured square (see Figure 6.1).

6.5 SUMMARY

This chapter described the development of a simple augmented reality application designed to test the capabilities of the prototype device. Differences between the 3D graphics environment on the embedded device and on a regular computer made it necessary to create a custom application, instead of merely porting one of the ARToolKit examples. The marker-tracking code used in the test application was copied directly from the ARToolKit source, without modification.

Different video-handling variations were provided, including alternative video capture back-ends and colourspace-conversion methods. Tests showed that the fastest option was to use the Video4Linux2 back-end and to perform colourspace conversion on the embedded graphics accelerator. This variant ran at a framerate of 7.5 fps when a marker was visible and 13.2 fps otherwise.

Further tests revealed that the most time-consuming parts of the AR processing pipeline were the marker pose calculation and background display tasks at around 40 ms each. The marker detection stage was not far behind, consuming 31 ms, while virtual object rendering required only 9 ms. Objects more complicated than the simple textured square used in the test application may be expected to take longer to render.

Apart from the hardware colourspace conversion, no attempt has been made to optimise the code for the device. Two features of the OMAP3530 processor offer

the potential for performance improvements. The ARM NEON floating point SIMD unit[4] may be useful in accelerating marker registration as that process relies heavily on floating-point operations. The on-board DSP may be employed in the marker detection stage, or colourspace conversion could be shifted to the DSP if the graphics accelerator is required for more complex rendering tasks.

[4] http://www.arm.com/products/processors/technologies/neon.php

Chapter 7

CONCLUSION

The final result of the project is an extendable platform designed to aid in the development of embedded augmented reality applications. This platform comprises an embedded computer with integrated camera, along with an operating system based on the open-source Linux kernel. An example program demonstrates how to access the device's capabilities in order to implement a typical augmented reality application.

The development platform uses the Gumstix Overo modular computing system. The Gumstix Overo computer offers a pre-built embedded system based around the OMAP3530 multimedia applications processor from Texas Instruments. Features offered by this system include a built-in graphics accelerator, high-definition video output and a camera interface, all of which are relevant to vision-based augmented reality.

One feature that the basic Gumstix system didn't provide was a camera to connect to the camera interface. A major part of the project was the development of a suitable camera peripheral, including creating the PCB and programming the drivers. With this peripheral, the device becomes a capable augmented reality platform.

Following this, an example augmented reality application was developed in order to demonstrate the device's capabilities. This application uses the marker-based approach, with marker-tracking routines drawn from the ARToolKit library. It runs at 8 to 13 frames per second, depending on whether or not a marker is in view.

Wherever possible, preexisting components and modules have been used in place of custom-developed solutions. The Gumstix Overo COM that forms the basis of the prototype device is one example; another is the set of camera ISP drivers originally developed for the Nokia N900. Although it took time to gain familiarity with these systems, this approach minimised the amount of new hardware and software that had to be developed.

Plans for future work on this device involve both modifications to the software and additions to the hardware. On the software side, performance may be improved by making better use of the OMAP3530 processor. This would consist of putting the on-board DSP to use and optimising the tracking code for the floating-point SIMD unit that is part of the ARM Cortex-A8 CPU.

At present, the prototype device is not completely wireless as it must be powered by a wall power supply and connected to a fixed monitor. These limitations may be removed by providing a battery and miniature display. One possibility is to make use of near-eye displays, such as those used in head-mounted display units, in order to offer a binocular-style form factor.

While interest in handheld augmented reality has grown steadily in recent years, most of this research is focussed on commercial smartphones and similar products. Compared to the device described in this paper, these products provide only limited access to the underlying hardware. It is hoped that this device will serve as a flexible development platform, allowing augmented reality researchers greater freedom to experiment with custom hardware and software.

Appendix A

OVEROCAM BOARD DESIGN

The Overocam camera board implements a simple circuit for interfacing the TCM8230MD VGA camera module with the OMAP3530 via the Gumstix Overo camera connector as described in Chapter 5. Figures A.1 and A.2 present the circuit schematic and board layout respectively. The parts list is given in Table A.1. The board uses two layers of copper and measures 35 mm $\times$ 30 mm.

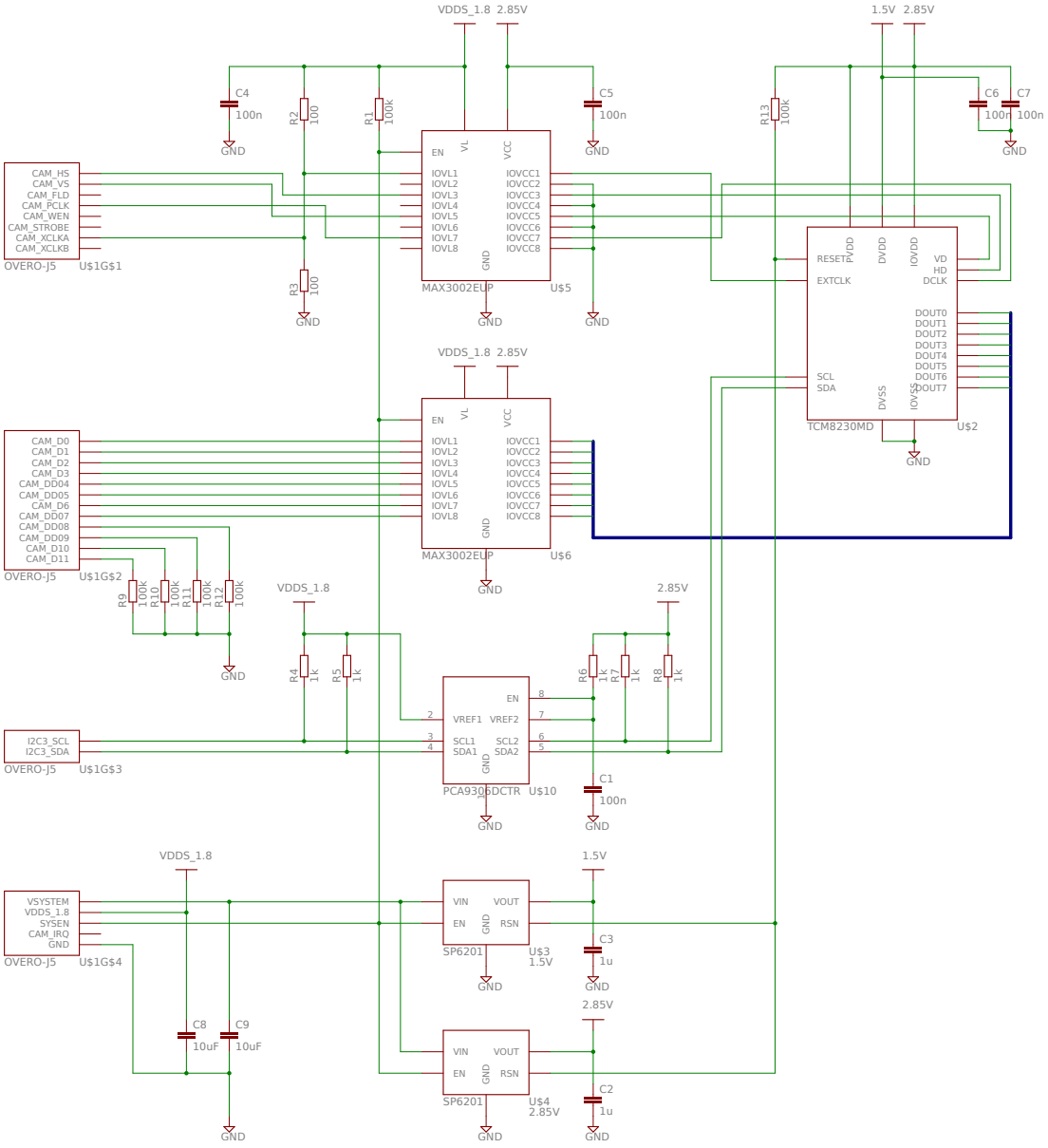


Figure A.1: Overocam circuit schematic.

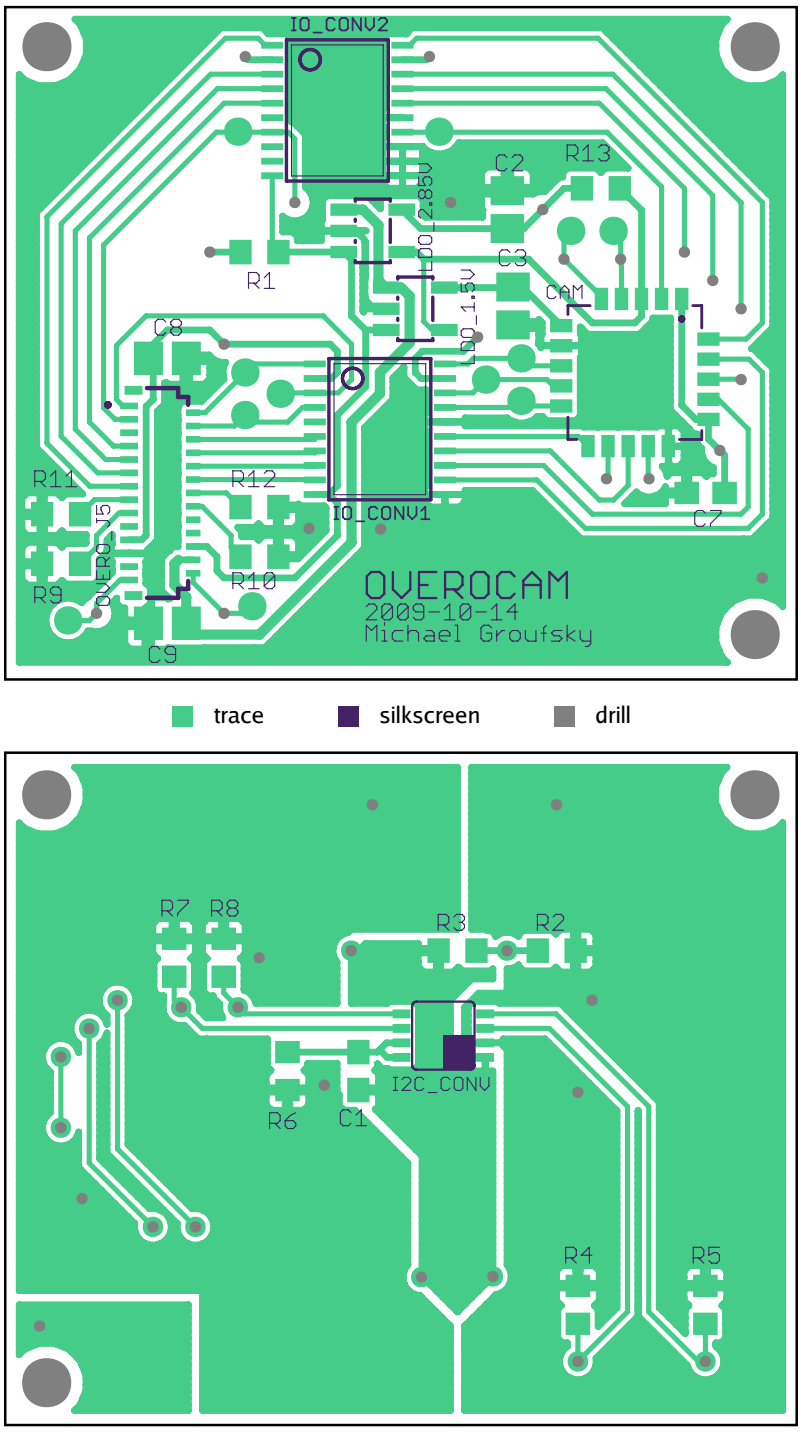


Figure A.2: Overocam board Layout, shown at 300% of actual size.

Part	Value	Description
OVERO_J5	FH26-27S-0.3SHW	27-pin Hirose connector
CAM	TCM8230MD	VGA camera module
IO_CONV1	MAX3002EUP	Logic level converter
IO_CONV2	MAX3002EUP	Logic level converter
I2C_CONV	PCA9306DCTR	I^2C level converter
LDO_1.5V	SP6201	LDO voltage regulator (1.5V)
LDO_2.85V	SP6201	LDO voltage regulator (2.85V)
R1, R9, R10, R11, R12, R13	100k	Resistor R0603
R2, R3	200	Resistor R0603
R4, R5, R6, R7, R8	1k	Resistor R0603
C1, C7	100n	Capacitor C0603
C2, C3	1u	Capacitor C0805
C8, C9	10u	Capacitor C0805

Table A.1: List of parts used in the Overocam circuit.

Appendix B

OVEROCAM DRIVER CODE

The kernel drivers for the Overocam board were developed starting from the main Gumstix Overo kernel tree, merging in the drivers from the OMAP3 camera ISP tree, then applying a number of custom additions and modifications. The code is too extensive to reproduce here, but a snapshot of the tree is made available on the CD accompanying this thesis in the directory linux-overocam.

These are the steps that were involved in the development process:

1. Clone the repository hosted at git://www.sakoman.com/git/linux-omap-2.6.git.
2. Add the "devel" branch of git://gitorious.org/omap3camera/mainline.git as a remote repository.
3. Crete a new local branch called "devel" and switch to it.
4. Merge the changes from the omap3camera repository.
5. Add the Overocam board definition file (commit 8f70020).
6. Add the driver for the TCM8230MD camera (commit 999103d).
7. Make the necessary changes to the OMAP3 camera ISP driver (commit c0c24a0).

The base repository that is used here is the "Linux 2.6 for OMAP" kernel hosted on sakoman.com[1]. This is the kernel that the Gumstix Overo computers are shipped with. The "devel" branch of omap3camera no longer exists, as development has moved to a different repository.

Notes on individual commits follow.

Commit 8f70020: Created board definition file for Overocam board.

- Modified: arch/arm/mach-omap2/Kconfig
- Modified: arch/arm/mach-omap2/Makefile

[1] http://www.sakoman.com/cgi-bin/gitweb.cgi?p=linux-omap-2.6.git

- New file: arch/arm/mach-omap2/board-overocam.c

Board configuration is defined in board-overocam.c. The modifications to Kconfig and Makefile introduce a new kernel config option, CONFIG_MACH_OVEROCAM. When activated, this causes the board-overocam module to be built.

Commit 999103d: Created TCM8230MD video device driver.

- Modified: drivers/media/video/Kconfig
- Modified: drivers/media/video/Makefile
- New file: drivers/media/video/tcm8230md.c
- New file: drivers/media/video/tcm8230md.h

The driver for the TCM8230MD camera module is implemented in tcm8230md.c. It provides minimal functionality, offering video in the YUV 4:2:2 format at 640×480 resolution and 30 frames per second only. The modifications to Kconfig and Makefile introduce a new kernel config option, CONFIG_VIDEO_TCM8230MD, which causes the tcm8230md module to be built.

Commit c0c24a0: Changes to OMAP3 camera ISP driver required for Overocam board.

- Modified: drivers/media/video/isp/isp.c
- Modified: drivers/media/video/isp/ispccdc.c
- Modified: drivers/media/video/isp/ispvideo.c
- Modified: drivers/media/video/omap34xxcam.c
- Modified: drivers/media/video/omap34xxcam.h

The most significant modifications involved configuring the ISP for YUV instead of RGB data. This necessitated various changes to ispccdc.c, ispvideo.c and omap34xxcam.c. In isp.c the code to enable and disable specific voltage supplies using the on-board regulators did not appear to be functioning and was removed. A minor change made to omap34xxcam.h reduced the number of expected image sensors from two to one.

Appendix C

AR TEST APPLICATION

The example AR application source code is spread across multiple files. The main files that define the program behaviour are reproduced below. These files are:

- ardemo.c: Program initialisation and main loop.
- ardemo_egl.c: EGL graphics context setup.
- ardemo_gl2_bg.c: Video frame background rendering.
- ardemo_gl2_vo.c: Virtual object rendering.
- ardemo_timer.c: Framerate reporting.
- ardemo_x11.c: Window setup and event handling.

In addition to these, a number of other files are required to compile the application, including:

- The video back-end definitions video-gst.c and video-v4l2.c.
- The ARToolKit marker detection and registration routines.
- The logo image data, in hitlabnz_logo.h.
- The ARToolKit header files.

These files have been omitted from this appendix for reasons of space, however they are available on the accompanying CD under the ardemo directory. This directory also contains Bitbake recipes for building the application.

For the application to run successfully, two additional files are required: camera_para.dat and pinwheel.patt. The first file contains generic camera parameters. The second file contains the marker pattern data that the application will look for. The marker pattern that is included in the files on the CD is shown in Figure C.1.

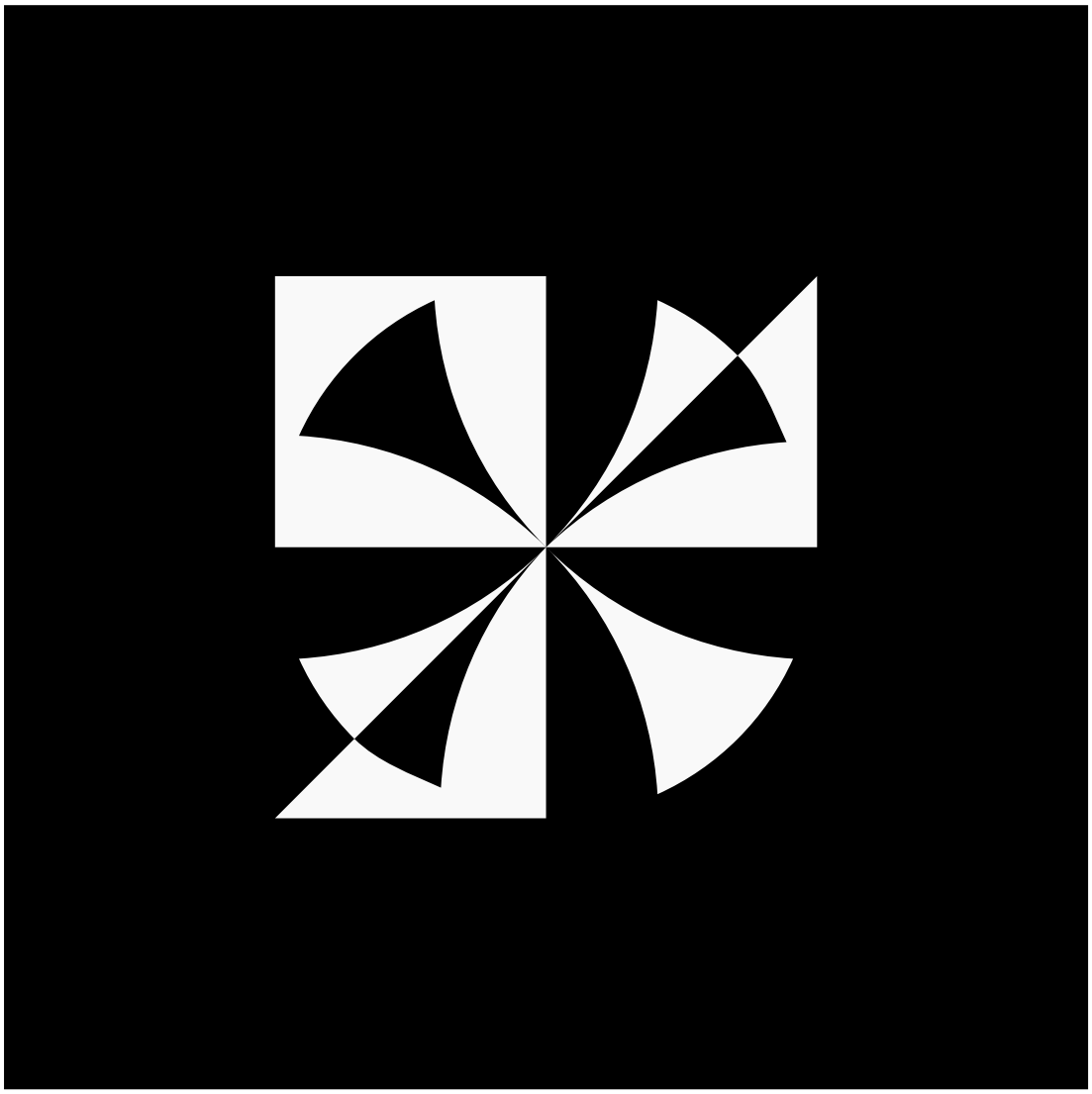


Figure C.1: The marker pattern used in the example application.

C.1 CODE LISTINGS

```
/*== ardemo.c: Program initialisation and main loop. ======================*/

#include <stdio.h>
#include <string.h>

#include <AR/video.h>
#include <AR/param.h>
#include <AR/ar.h>

/* Pixel intensity threshold used to binarise image. */
#define THRESHOLD 128

/* Datafile containing camera parameters. */
#define PARAMDATA "Data/camera_para.dat"

/* Datafile containing marker pattern. */
#define PATTDATA "Data/pinwheel.patt"

/* Defined in ardemo_x11.c */
extern int initDisplay(int, int);
extern int handleEvents(void);

/* Defined in ardemo_egl.c */
extern void flip(void);

/* Defined in ardemo_gl2_bg.c */
extern void initBackground(int, int);
extern int loadImageData(unsigned char *);
extern int drawBackground(void);

/* Defined in ardemo_gl2_vo.c */
extern void initVirtualObject(double *, double *, int, int);
extern int drawVirtualObject(double *);

/* Defined in ardemo_gl2_vo.c */
extern void startTimer(void);
extern void checkTimer(void);

/* Variables used to control execution of different stages. */
int detection_enable = 1;
int registration_enable = 1;
int augmentation_enable = 1;
int background_enable = 1;

/* Search the video frame for markers and return the one that best matches
the given marker pattern. */
ARMarkerInfo *findMarker(ARUint8 *imgData, int patt_id) {
        ARMarkerInfo *marker_list;
        ARMarkerInfo *marker_best = NULL;
        int n, i;

        if (arDetectMarker(imgData, THRESHOLD, &marker_list, &n) == -1)
                return NULL;

        for (i = 0; i < n; i++)
```

```
                if (marker_list[i].id == patt_id)
                        if (marker_best == NULL
                            || marker_best->cf < marker_list[i].cf)
                                marker_best = &marker_list[i];

        return marker_best;
}

/* Calculate the marker position and orientation and use this info to
render the virtual object. */
void showObject(ARMarkerInfo *marker_info) {
        double patt_width = 80.0;
        double patt_center[] = { 0.0, 0.0 };
        double patt_trans[3][4];

        arGetTransMat(marker_info, patt_center, patt_width, patt_trans);

        if (augmentation_enable)
                drawVirtualObject((double *)patt_trans);
}

/* Load the camera parameters from PARAMDATA. */
int loadCameraParam(int x, int y) {
        ARParam param;
        double icpara[3][4];
        double trans[3][4];

        if (arParamLoad(PARAMDATA, 1, &param) == -1) return 1;

        arParamChangeSize(&param, x, y, &param);
        arInitCparam(&param);

        arParamDecompMat(param.mat, icpara, trans);
        initVirtualObject((double *)icpara, (double *)trans,
          param.xsize, param.ysize);

        return 0;
}

int main(int argc, char *argv[]) {
        int i;
        int width, height;
        int patt_id;
        enum { SUCCESS,
               ERR_USAGE,
               ERR_VIDEODEV,
               ERR_GFXINIT,
               ERR_PATTERN,
               ERR_CAMPARAM,
               ERR_VIDEOCAP } errlvl = SUCCESS;

        /* Process command line arguments. */
        for (i = 1; i < argc; i++) {
                if (strncmp(argv[i], "--no-detection", 8) == 0) {
                        detection_enable = 0;
                        continue;
                }
```

```
        if (strncmp(argv[i], "--no-registration", 8) == 0) {
                registration_enable = 0;
                continue;
        }
        if (strncmp(argv[i], "--no-augmentation", 8) == 0) {
                augmentation_enable = 0;
                continue;
        }
        if (strncmp(argv[i], "--no-background", 8) == 0) {
                background_enable = 0;
                continue;
        }
        printf("Usage: %s [--no-detection] [--no-registration] "
          "[--no-augmentation]\n", argv[0]);
        errlvl = ERR_USAGE;
        goto cleanup;
}

/* Initialise application. */

if (arVideoOpen(NULL) != 0) {
        fprintf(stderr, "Failed to open video device.\n");
        errlvl = ERR_VIDEODEV;
        goto cleanup;
}

arVideoInqSize(&width, &height);

if (initDisplay(width, height)) {
        fprintf(stderr, "Failed to initialise graphics display.\n");
        errlvl = ERR_GFXINIT;
        goto cleanup;
}

if (loadCameraParam(width, height)) {
        fprintf(stderr, "Failed to load camera parameters.\n");
        errlvl = ERR_CAMPARAM;
        goto cleanup;
}

initBackground(width, height);

patt_id = arLoadPatt(PATTDATA);

if (patt_id == -1) {
        fprintf(stderr, "Failed to load marker pattern.\n");
        errlvl = ERR_PATTERN;
        goto cleanup;
}

if (arVideoCapStart() != 0) {
        fprintf(stderr, "Failed to start video capture.\n");
        errlvl = ERR_VIDEOCAP;
        goto cleanup;
}

/* Main processing loop.  Exit when a key is pressed. */
```

```
        for (startTimer(); handleEvents() == 0; checkTimer()) {
                ARMarkerInfo *marker_info = NULL;
                ARUint8 *imgData;

                imgData = arVideoGetImage();

                if (imgData == NULL) continue;

                if (detection_enable)
                        marker_info = findMarker(imgData, patt_id);

                if (background_enable)
                        loadImageData(imgData);

                arVideoCapNext();

                if (background_enable)
                        drawBackground();

                if (registration_enable && marker_info != NULL) {
                        const double patt_width = 80.0;
                        const double patt_center[] = { 0.0, 0.0 };
                        double patt_trans[3][4];

                        arGetTransMat(marker_info, patt_center, patt_width,
                          patt_trans);

                        if (augmentation_enable)
                                drawVirtualObject((double *)patt_trans);
                }

                flip();
        }

cleanup:
        arVideoCapStop();
        arVideoClose();

        return errlvl;
}


/*== ardemo_x11.c: Window setup and event handling. ========================*/

#include <stdio.h>
#include <assert.h>

#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/keysymdef.h>

#include <EGL/egl.h>

/* Defined in ardemo_egl.c */
extern int setupEGL(EGLNativeDisplayType, EGLNativeWindowType);

Display *display = NULL;
```

```
static int createColormap(int screen_number, Window w, Colormap *colormap) {
        XVisualInfo vinfo;
        int depth;

        depth = DefaultDepth(display, screen_number);

        if (!XMatchVisualInfo(display, screen_number, depth, TrueColor,
          &vinfo)) return 1;

        *colormap = XCreateColormap(display, w, vinfo.visual, AllocNone);

        return 0;
}

static int createWindow(int width, int height, Window *window) {
        Window parent;
        unsigned long valuemask;
        XSetWindowAttributes attributes;
        int screen_number;

        screen_number = DefaultScreen(display);
        parent = RootWindow(display, screen_number);
        valuemask = CWBackPixel | CWBorderPixel | CWEventMask | CWColormap;
        attributes.event_mask = StructureNotifyMask
                              | ExposureMask
                              | KeyPressMask;

        if (createColormap(screen_number, parent, &(attributes.colormap)))
                return 1;

        *window = XCreateWindow(display, parent, 0, 0, width, height, 0,
          CopyFromParent, InputOutput, CopyFromParent, valuemask, &attributes);

        return 0;
}

/* Initialise the display and create the main application window. */
int initDisplay(int width, int height) {
        Window window;

        display = XOpenDisplay(0);
        if (display == NULL) {
                fprintf(stderr, "Could␣not␣open␣X11␣display.\n");
                return 1;
        }

        if (createWindow(width, height, &window)) {
                fprintf(stderr, "Could␣not␣create␣window.\n");
                return 1;
        }

        XMapWindow(display, window);
        XFlush(display);

        if (setupEGL((EGLNativeDisplayType)display,
          (EGLNativeWindowType)window)) return 1;
```

```
        return 0;
}

/* Handle pending X11 events.  Return nonzero if the application window is
closed or any key is pressed. */
int handleEvents(void) {
        XEvent event;

        assert(display != NULL);

        XFlush(display);

        if (XPending(display) == 0) return 0;

        XNextEvent(display, &event);

        switch (event.type) {
                KeyCode keycode;
        case DestroyNotify:
                return 1;
        case KeyPress:
                keycode = ((XKeyEvent *)&event)->keycode;
                switch (XKeycodeToKeysym(display, keycode, 0)) {
                case XK_Escape:
                        return 1;
                }
        }

        return 0;
}


/*== ardemo_egl.c: EGL graphics context setup. ============================*/

#include <stdio.h>

#include <EGL/egl.h>

EGLDisplay eglDisplay;
EGLSurface eglSurface;

static int createContext(EGLConfig eglConfig, EGLContext *eglContext) {
        EGLint const attrs[] = { EGL_CONTEXT_CLIENT_VERSION, 2,
                                 EGL_NONE };

        *eglContext = eglCreateContext(eglDisplay, eglConfig, NULL, attrs);

        return 0;
}

static int chooseConfig(EGLConfig *eglConfig) {
        EGLint const attrs[] = { EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
                                 EGL_RENDERABLE_TYPE, EGL_OPENGL_ES2_BIT,
                                 EGL_NONE };
        int num_config;

        if (!eglChooseConfigeglDisplay, attrs, eglConfig, 1, &num_config))
                return 1;
```

```
        if (num_config != 1) return 1;

        return 0;
}

/* Set up an OpenGL rendering context using EGL. */
int setupEGL(EGLNativeDisplayType display, EGLNativeWindowType window) {
        EGLConfig eglConfig;
        EGLContext eglContext;

        eglDisplay = eglGetDisplay(display);

        if (eglDisplay == EGL_NO_DISPLAY) {
                fprintf(stderr, "Could␣not␣get␣EGL␣display.\n");
                return 1;
        }

        if (!eglInitialize(eglDisplay, NULL, NULL)) {
                fprintf(stderr, "Could␣not␣initialise␣EGL␣display.\n");
                return 1;
        }

        eglBindAPI(EGL_OPENGL_ES_API);

        if (chooseConfig(&eglConfig)) {
                fprintf(stderr, "Could␣not␣select␣EGL␣configuration.\n");
                return 1;
        }

        eglSurface = eglCreateWindowSurface(eglDisplay, eglConfig, window,
          NULL);

        createContext(eglConfig, &eglContext);

        eglMakeCurrent(eglDisplay, eglSurface, eglSurface, eglContext);

        return 0;
}

/* Flip the graphics buffers; i.e., update the display. */
void flip(void) {
        eglSwapBuffers(eglDisplay, eglSurface);
}


/*== ardemo_gl2_bg.c: Video frame background rendering. ====================*/

#include <stdlib.h>
#include <time.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>

#include <GLES2/gl2.h>

#include <AR/config.h>
```

```
/* If SOFTYUV2RGB is defined, colourspace conversion will be performed in
software instead of on the GPU.  The only point of this is to compare the
difference in performance. */
/* This has an effect only if the YUV pixel format is used. */
//#define SOFTYUV2RGB

/* Vertex shader attribute locations. */
enum { ATTR_POSITION, ATTR_TEXCOORD };

#ifdef SOFTYUV2RGB
static GLubyte *pixeldata;
#endif
/* Background width and height, in pixels. */
static int width, height;

/* Shader program used to render the background. */
static GLuint programObject;

void loadImageData(unsigned char *imgdata) {

#if (AR_DEFAULT_PIXEL_FORMAT == AR_PIXEL_FORMAT_RGB)
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
          GL_UNSIGNED_BYTE, imgdata);
#elif (AR_DEFAULT_PIXEL_FORMAT == AR_PIXEL_FORMAT_YUY2)
#ifdef SOFTYUV2RGB
        enum { R0, G0, B0, R1, G1, B1, RGB_BPP = 3 };
        enum { Y0, U, Y1, V, YUV_BPP = 2 };
        int i;

        assert(pixeldata != NULL);

        for (i = 0; i < width * height; i += 2) {
                GLubyte *pixel;
                unsigned char *imgpt;
                short chroma_r, chroma_g, chroma_b;
                short luma_0, luma_1;

                pixel = &pixeldata[i*RGB_BPP];
                imgpt = &imgdata[i*YUV_BPP];

                chroma_r = 2 * imgpt[V] - 256;
                chroma_g = 256 - imgpt[V] - imgpt[U];
                chroma_b = 2 * imgpt[U] - 256;

                luma_0 = imgpt[Y0];
                luma_1 = imgpt[Y1];

#define clip(X) ((X) <= 0 ? 0 : (X) >= 255 ? 255 : (X))
                pixel[R0] = clip(luma_0 + chroma_r);
                pixel[G0] = clip(luma_0 + chroma_g);
                pixel[B0] = clip(luma_0 + chroma_b);
                pixel[R1] = clip(luma_1 + chroma_r);
                pixel[G1] = clip(luma_1 + chroma_g);
                pixel[B1] = clip(luma_1 + chroma_b);
#undef clip
        }
```

```
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
          GL_UNSIGNED_BYTE, pixeldata);
#else
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width / 2, height, 0, GL_RGBA,
          GL_UNSIGNED_BYTE, imgdata);
#endif
#else
#error Unsupported pixel format.
#endif
}

static int compileFragShader(GLuint *fragShader) {
        const char *fragShaderStr =
                "precision␣mediump␣float;\n"
                "varying␣vec2␣v_texCoord;\n"
                "uniform␣sampler2D␣s_texture;\n"
#if (AR_DEFAULT_PIXEL_FORMAT == AR_PIXEL_FORMAT_RGB)
                "void␣main()\n"
                "{\n"
                "␣␣gl_FragColor␣=␣texture2D(s_texture,␣v_texCoord);\n"
                "}\n";
#elif (AR_DEFAULT_PIXEL_FORMAT == AR_PIXEL_FORMAT_YUY2)
#ifdef SOFTYUV2RGB
                "void␣main()\n"
                "{\n"
                "␣␣gl_FragColor␣=␣texture2D(s_texture,␣v_texCoord);\n"
                "}\n";
#else
        /* Note that this conversion method averages the two y-values in
        each YUYV quartet, so the horizontal resolution is halved. */
                "const␣mat4␣yuv2rgb_a␣=␣mat4(␣0.5,␣␣0.5,␣␣0.5,␣0.0,\n"
                "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣0.0,␣-1.0,␣␣2.0,␣0.0,\n"
                "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣0.5,␣␣0.5,␣␣0.5,␣0.0,\n"
                "␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣2.0,␣-1.0,␣␣0.0,␣0.0);\n"
                "const␣vec4␣yuv2rgb_k␣=␣vec4(-1.0,␣␣1.0,␣-1.0,␣1.0);\n"
                "void␣main()\n"
                "{\n"
                "␣␣gl_FragColor␣=␣yuv2rgb_a\n"
                "␣␣␣␣*␣texture2D(s_texture,␣v_texCoord)␣+␣yuv2rgb_k;\n"
                "}\n";
#endif
#else
#error Unsupported pixel format.
#endif
        GLint shaderCompiled;

        *fragShader = glCreateShader(GL_FRAGMENT_SHADER);
        glShaderSource(*fragShader, 1, &fragShaderStr, NULL);
        glCompileShader(*fragShader);
        glGetShaderiv(*fragShader, GL_COMPILE_STATUS, &shaderCompiled);

        assert(shaderCompiled);

        return 0;
}

static int compileVertShader(GLuint *vertShader) {
```

```
        const char *vertShaderStr =
                "attribute␣vec4␣a_Position;\n"
                "attribute␣vec2␣a_texCoord;\n"
                "varying␣vec2␣v_texCoord;\n"
                "void␣main()\n"
                "{\n"
                "␣␣␣gl_Position␣=␣a_Position;\n"
                "␣␣␣v_texCoord␣=␣a_texCoord;\n"
                "}\n";
        GLint shaderCompiled;

        *vertShader = glCreateShader(GL_VERTEX_SHADER);
        glShaderSource(*vertShader, 1, &vertShaderStr, NULL);
        glCompileShader(*vertShader);
        glGetShaderiv(*vertShader, GL_COMPILE_STATUS, &shaderCompiled);

        assert(shaderCompiled);

        return 0;
}

static int linkProgram(GLuint fragShader, GLuint vertShader) {
        GLint programLinked;

        programObject = glCreateProgram();
        glAttachShader(programObject, fragShader);
        glAttachShader(programObject, vertShader);

        glBindAttribLocation(programObject, ATTR_POSITION, "a_Position");
        glBindAttribLocation(programObject, ATTR_TEXCOORD, "a_texCoord");
        glLinkProgram(programObject);
        glGetProgramiv(programObject, GL_LINK_STATUS, &programLinked);

        assert(programLinked);

        return 0;
}

static void createTexture(void) {
        GLuint textureId;

        glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

        glGenTextures(1, &textureId);

        glBindTexture(GL_TEXTURE_2D, textureId);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
}

void initBackground(int w, int h) {
        GLuint fragShader, vertShader;

        compileFragShader(&fragShader);
        compileVertShader(&vertShader);
        linkProgram(fragShader, vertShader);
```

```
        createTexture();
        width = w;
        height = h;
#ifdef SOFTYUV2RGB
        pixeldata = malloc(w * h * 3);
#endif
}

/*
Background coords:

  (X0, Y1)+-------------+(X1, Y1)
          |             |
          |             |
          |             |
          |             |
  (X0, Y0)+-------------+(X1, Y0)

  (Depth Z)

*/

#define X0 -1.0f
#define X1 +1.0f
#define Y0 -1.0f
#define Y1 +1.0f
#define Z   1.0f

#define U0 0.0f
#define U1 1.0f
#define V0 1.0f
#define V1 0.0f

/* Draw the background, consisting of a textured square showing the video
frame. */
void drawBackground(void) {
        GLfloat vertices[] = { X0, Y0, Z, U0, V0,
                               X0, Y1, Z, U0, V1,
                               X1, Y0, Z, U1, V0,
                               X1, Y1, Z, U1, V1 };

        glUseProgram(programObject);

        glVertexAttribPointer(ATTR_POSITION, 3, GL_FLOAT, GL_FALSE,
          5 * sizeof (GLfloat), vertices);
        glVertexAttribPointer(ATTR_TEXCOORD, 2, GL_FLOAT, GL_FALSE,
          5 * sizeof (GLfloat), vertices + 3);

        glEnableVertexAttribArray(ATTR_POSITION);
        glEnableVertexAttribArray(ATTR_TEXCOORD);

        glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

        glDisableVertexAttribArray(ATTR_POSITION);
        glDisableVertexAttribArray(ATTR_TEXCOORD);
}
```

```
/*== ardemo_gl2_vo.c: Virtual object rendering. ============================*/

#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <assert.h>

#include <GLES2/gl2.h>

/* Logo image data */
#include "hitlabnz_logo.h"

/* Vertex shader attribute locations. */
enum { ATTR_POSITION, ATTR_TEXCOORD };

/* Shader program used to render the virtual object. */
static GLuint programObject;

static GLfloat m_projection[4][4];
static GLfloat m_modelview[4][4];

/* The algorithm for converting from artk camera parameters to an ogl matrix
is taken from gsub-lite. */
#define VIEW_DISTANCE_MIN   0.1
#define VIEW_DISTANCE_MAX 100.0
static void calculateProjection(double icpara[3][4], double trans[3][4],
  int width, int height) {
        int i, j;
        double p[3][3], q[4][4] = { 0 };

        for (i = 0; i < 4; i++)
                icpara[1][i] = (height - 1) * icpara[2][i] - icpara[1][i];

        for (i = 0; i < 3; i++)
                for(j = 0; j < 3; j++)
                        p[i][j] = icpara[i][j] / icpara[2][2];

        q[0][0] =  (2.0 * p[0][0]) / (width - 1);
        q[0][1] =  (2.0 * p[0][1]) / (width - 1);
        q[0][2] = -(2.0 * p[0][2]) / (width - 1) + 1.0;

        q[1][1] = -(2.0 * p[1][1]) / (height - 1);
        q[1][2] = -(2.0 * p[1][2]) / (height - 1) + 1.0;

        q[2][2] = (VIEW_DISTANCE_MIN + VIEW_DISTANCE_MAX) /
          (VIEW_DISTANCE_MIN - VIEW_DISTANCE_MAX);
        q[2][3] = 2.0 * VIEW_DISTANCE_MIN * VIEW_DISTANCE_MAX /
          (VIEW_DISTANCE_MIN - VIEW_DISTANCE_MAX);

        q[3][2] = -1.0;

        for (i = 0; i < 4; i++) {
                for (j = 0; j < 3; j++) {
                        m_projection[j][i] = q[i][0] * trans[0][j] +
                                             q[i][1] * trans[1][j] +
                                             q[i][2] * trans[2][j];
                m_projection[3][i] = q[i][0] * trans[0][3] +
```

```
                                        q[i][1] * trans[1][3] +
                                        q[i][2] * trans[2][3] +
                                        q[i][3];
                }
        }
}
#undef VIEW_DISTANCE_MIN
#undef VIEW_DISTANCE_MAX

/* The algorithm for converting from artk trans. matrix to an ogl matrix is
taken from gsub-lite. */
#define VIEW_SCALEFACTOR 0.025
static void calculateModelview(double trans[3][4]) {
        m_modelview[0][0] =  trans[0][0];
        m_modelview[0][1] = -trans[1][0];
        m_modelview[0][2] = -trans[2][0];
        m_modelview[0][3] = 0.0;
        m_modelview[1][0] =  trans[0][1];
        m_modelview[1][1] = -trans[1][1];
        m_modelview[1][2] = -trans[2][1];
        m_modelview[1][3] = 0.0;
        m_modelview[2][0] =  trans[0][2];
        m_modelview[2][1] = -trans[1][2];
        m_modelview[2][2] = -trans[2][2];
        m_modelview[2][3] = 0.0;
        m_modelview[3][0] = VIEW_SCALEFACTOR *  trans[0][3];
        m_modelview[3][1] = VIEW_SCALEFACTOR * -trans[1][3];
        m_modelview[3][2] = VIEW_SCALEFACTOR * -trans[2][3];
        m_modelview[3][3] = 1.0;
}
#undef VIEW_SCALEFACTOR

static void applyMVP(void) {
        GLint location;
        GLfloat matrix[4][4] = { 0.0 };
        int i, j, k;

        for (i = 0; i < 4; i++)
                for (j = 0; j < 4; j++)
                        for (k = 0; k < 4; k++)
                                matrix[i][j]
                                  += m_modelview[i][k] * m_projection[k][j];

        location = glGetUniformLocation(programObject, "mvp");
        glUniformMatrix4fv(location, 1, GL_FALSE, (GLfloat *)matrix);
}

static int compileFragShader(GLuint *fragShader) {
        const char *fragShaderStr =
                "precision␣mediump␣float;\n"
                "varying␣vec2␣v_texCoord;\n"
                "uniform␣sampler2D␣s_texture;\n"
                "void␣main()\n"
                "{\n"
                "␣␣gl_FragColor␣=␣texture2D(s_texture,␣v_texCoord);\n"
                "}\n";
        GLint shaderCompiled;
```

```
        *fragShader = glCreateShader(GL_FRAGMENT_SHADER);
        glShaderSource(*fragShader, 1, &fragShaderStr, NULL);
        glCompileShader(*fragShader);
        glGetShaderiv(*fragShader, GL_COMPILE_STATUS, &shaderCompiled);

        assert(shaderCompiled);

        return 0;
}

static int compileVertShader(GLuint *vertShader) {
        const char *vertShaderStr =
                "attribute␣vec4␣a_Position;\n"
                "attribute␣vec2␣a_texCoord;\n"
                "varying␣vec2␣v_texCoord;\n"
                "uniform␣mat4␣mvp;\n"
                "void␣main()\n"
                "{\n"
                "␣␣␣gl_Position␣=␣mvp␣*␣a_Position;\n"
                "␣␣␣v_texCoord␣=␣a_texCoord;\n"
                "}\n";
        GLint shaderCompiled;

        *vertShader = glCreateShader(GL_VERTEX_SHADER);
        glShaderSource(*vertShader, 1, &vertShaderStr, NULL);
        glCompileShader(*vertShader);
        glGetShaderiv(*vertShader, GL_COMPILE_STATUS, &shaderCompiled);

        assert(shaderCompiled);

        return 0;
}

static int linkProgram(GLuint fragShader, GLuint vertShader) {
        GLint programLinked;

        programObject = glCreateProgram();
        glAttachShader(programObject, fragShader);
        glAttachShader(programObject, vertShader);

        glBindAttribLocation(programObject, ATTR_POSITION, "a_Position");
        glBindAttribLocation(programObject, ATTR_TEXCOORD, "a_texCoord");
        glLinkProgram(programObject);
        glGetProgramiv(programObject, GL_LINK_STATUS, &programLinked);

        assert(programLinked);

        return 0;
}

static void createTexture(void) {
        GLuint textureId;

        glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

        glGenTextures(1, &textureId);
```

```
        glBindTexture(GL_TEXTURE_2D, textureId);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
}

void initVirtualObject(double icpara[3][4], double trans[3][4], int width,
  int height) {
        GLuint fragShader, vertShader;

        compileFragShader(&fragShader);
        compileVertShader(&vertShader);
        linkProgram(fragShader, vertShader);

        createTexture();

        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

        calculateProjection(icpara, trans, width, height);
}

/* Draw the virtual object, consisting of a textured square spinning in
place. */
void drawVirtualObject(double patt_trans[3][4]) {
        enum { X, Y, Z, U, V, NCOMP };
        GLfloat vertices[4][NCOMP];
        struct timespec time;
        double theta, p, q;

        clock_gettime(CLOCK_REALTIME, &time);
        theta = 2 * M_PI * (time.tv_nsec * 1e-9);
        p = cos(theta);
        q = sin(theta);

        /* Geometry coords. */
        vertices[0][X] = vertices[1][X] = +p;
        vertices[2][X] = vertices[3][X] = -p;
        vertices[0][Y] = vertices[1][Y] = +q;
        vertices[2][Y] = vertices[3][Y] = -q;
        vertices[0][Z] = vertices[2][Z] = 0.0;
        vertices[1][Z] = vertices[3][Z] = 2.0;

        /* Texture coords. */
        vertices[0][U] = vertices[1][U] = 0.0f;
        vertices[2][U] = vertices[3][U] = 1.0f;
        vertices[0][V] = vertices[2][V] = 1.0f;
        vertices[1][V] = vertices[3][V] = 0.0f;

        glUseProgram(programObject);

        calculateModelview(patt_trans);
        applyMVP();

        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, HITLABNZ_LOGO_WIDTH,
          HITLABNZ_LOGO_HEIGHT, 0, GL_RGBA, GL_UNSIGNED_BYTE,
```

```
          HITLABNZ_LOGO_PIXEL_DATA);

        glVertexAttribPointer(ATTR_POSITION, 3, GL_FLOAT, GL_FALSE,
          NCOMP * sizeof (GLfloat), vertices);
        glVertexAttribPointer(ATTR_TEXCOORD, 2, GL_FLOAT, GL_FALSE,
          NCOMP * sizeof (GLfloat), (GLfloat *) vertices + 3);

        glEnableVertexAttribArray(ATTR_POSITION);
        glEnableVertexAttribArray(ATTR_TEXCOORD);

        glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

        glDisableVertexAttribArray(ATTR_POSITION);
        glDisableVertexAttribArray(ATTR_TEXCOORD);
}

/*== ardemo_timer.c: Framerate reporting. ================================*/

#include <stdio.h>
#include <time.h>

/* Used to track the real time elapsed. */
struct timespec start, end;

/* Used to track the number of frames elapsed in the last second. */
int framecount = 0;

/* Initialise the start time. */
void startTimer(void) {
        clock_gettime(CLOCK_REALTIME, &start);
}

/* Print out the current framerate, but not more than once per second.
Should be called exactly once each frame. */
void checkTimer(void) {
        framecount++;

        clock_gettime(CLOCK_REALTIME, &end);

        if (end.tv_sec > start.tv_sec) {
                float seconds = end.tv_sec - start.tv_sec
                              + (end.tv_nsec - start.tv_nsec) * 1e-9;

                printf("Framerate:␣%f␣fps␣(%f␣ms␣per␣frame)\n",
                  framecount / seconds, (seconds / framecount) * 1e3);

                framecount = 0;

                /* Restart the timer. */
                clock_gettime(CLOCK_REALTIME, &start);
        }
}
```

REFERENCES

Ashley, S. (2008), 'Annotating the real world', *Scientific American*, Vol. 299, No. 4, pp. 27–28.

Azuma, R.T. (1997), 'A survey of augmented reality', *Presence: Teleoperators and Virtual Environments*, Vol. 6, No. 4, August, pp. 355–385.

Azuma, R., Hoff, B., Neely, H., I. and Sarfaty, R. (1999), 'A motion-stabilized outdoor augmented reality system', In *Virtual Reality, 1999. Proceedings., IEEE*, March, pp. 252–259.

Azuma, R., Baillot, Y., Behringer, R., Feiner, S., Julier, S. and MacIntyre, B. (2001), 'Recent advances in augmented reality', *Computer Graphics and Applications, IEEE*, Vol. 21, No. 6, November, pp. 34–47.

Bajura, M. and Neumann, U. (1995), 'Dynamic registration correction in video-based augmented reality systems', *Computer Graphics and Applications, IEEE*, Vol. 15, No. 5, September, pp. 52–60.

Baumgartner, D., Roessler, P., Kubinger, W., Zinner, C. and Ambrosch, K. (2009), 'Benchmarks of low-level vision algorithms for DSP, FPGA, and mobile PC processors', In Kisačanin, B., Bhattacharyya, S.S. and Chai, S. (Eds.), *Embedded Computer Vision*, Springer-Verlag, London, UK, Chap. 5, pp. 101–120.

Billinghurst, M., Kato, H. and Poupyrev, I. (2001), 'MagicBook: transitioning between reality and virtuality', In *CHI '01: CHI '01 extended abstracts on Human factors in computing systems*, ACM, New York, NY, USA, pp. 25–26.

Bleser, G., Wuest, H. and Strieker, D. (2006), 'Online camera pose estimation in partially known and dynamic scenes', In *Mixed and Augmented Reality, 2006. ISMAR 2006. IEEE/ACM International Symposium on*, October, pp. 56–65.

Caarls, W., Jonker, P. and Corporaal, H. (2002), 'SmartCam: Devices for embedded intelligent cameras', In Schweizer, M. (Ed.), *Proc. 3rd PROGRESS Workshop on Embedded Systems*, Utrecht, Netherlands, October.

Caudell, T. and Mizell, D. (1992), 'Augmented reality: an application of heads-up display technology to manual manufacturing processes', In *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, January, pp. 659–669.

Chaoui, J., Cyr, K., de Gregorio, S., Giacalone, J.P., Webb, J. and Masse, Y. (2001), 'Open Multimedia Application Platform: enabling multimedia applications in third generation wireless terminals through a combined RISC/DSP architecture', In *Acoustics, Speech, and Signal Processing, 2001. Proceedings. (ICASSP '01). 2001 IEEE International Conference on*, pp. 1009–1012.

Chen, W.C., Xiong, Y., Gao, J., Gelfand, N. and Grzeszczuk, R. (2007), 'Efficient extraction of robust image features on mobile devices', In *Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on*, November, pp. 287–288.

Cho, Y., Lee, J. and Neumann, U. (1998), 'A multi-ring color fiducial system and an intensity-invariant detection method for scalable fiducial-tracking augmented reality', In *IWAR '98*, pp. 147–165.

Comport, A., Marchand, E. and Chaumette, F. (2003), 'A real-time tracker for markerless augmented reality', In *Mixed and Augmented Reality, 2003. Proceedings. The Second IEEE and ACM International Symposium on*, October, pp. 36–45.

Comport, A., Marchand, E., Pressigout, M. and Chaumette, F. (2006), 'Real-time markerless tracking for augmented reality: the virtual visual servoing framework', *Visualization and Computer Graphics, IEEE Transactions on*, Vol. 12, No. 4, July, pp. 615–628.

Davison, A., Mayol, W. and Murray, D. (2003), 'Real-time localization and mapping with wearable active vision', In *Mixed and Augmented Reality, 2003. Proceedings. The Second IEEE and ACM International Symposium on*, October, pp. 18–27.

Durrant-Whyte, H. and Bailey, T. (2006), 'Simultaneous localization and mapping: part I', *Robotics & Automation Magazine, IEEE*, Vol. 13, No. 2, June, pp. 99–110.

Feiner, S., Macintyre, B. and Seligmann, D. (1993), 'Knowledge-based augmented reality', *Commun. ACM*, Vol. 36, No. 7, pp. 53–62.

Feiner, S., MacIntyre, B., Hollerer, T. and Webster, A. (1997), 'A touring machine: Prototyping 3D mobile augmented reality systems for exploring the

urban environment', In *ISWC '97: Proceedings of the 1st IEEE International Symposium on Wearable Computers*, IEEE Computer Society, Washington, DC, USA, p. 74.

Fiala, M. (2005a), 'ARTag, a fiducial marker system using digital techniques', In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, June, pp. 590–596.

Fiala, M. (2005b), 'Comparing ARTag and ARToolkit Plus fiducial marker systems', In *Haptic Audio Visual Environments and their Applications, 2005. IEEE International Workshop on*, October.

Fischler, M.A. and Bolles, R.C. (1981), 'Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography', *Commun. ACM*, Vol. 24, No. 6, pp. 381–395.

Fürtler, J., Rössler, P., Brodersen, J., Nachtnebel, H., Mayer, K.J., Cadek, G. and Eckel, C. (2007), 'Design considerations for scalable high-performance vision systems embedded in industrial print inspection machines', *EURASIP J. Embedded Syst.*, Vol. 2007, January, pp. 30–30.

Gausemeier, J., Fruend, J., Matysczok, C., Bruederlin, B. and Beier, D. (2003), 'Development of a real time image based object recognition method for mobile ar-devices', In *AFRIGRAPH '03: Proceedings of the 2nd international conference on Computer graphics, virtual Reality, visualisation and interaction in Africa*, ACM, New York, NY, USA, pp. 133–139.

Genc, Y., Riedel, S., Souvannavong, F., Akinlar, C. and Navab, N. (2002), 'Marker-less tracking for AR: a learning-based approach', In *Mixed and Augmented Reality, 2002. ISMAR 2002. Proceedings. International Symposium on*, pp. 295–304.

Guimarães, G.F., Lima, J.P.S.M., Teixeira, J.M.X.N., Silva, G.D., Teichrieb, V. and Kelner, J. (2007), 'FPGA infrastructure for the development of augmented reality applications', In *SBCCI '07: Proceedings of the 20th annual conference on Integrated circuits and systems design*, ACM, New York, NY, USA, pp. 336–341.

Henrysson, A. and Ollila, M. (2004), 'UMAR: Ubiquitous mobile augmented reality', In *MUM '04: Proceedings of the 3rd international conference on Mobile and ubiquitous multimedia*, ACM, New York, NY, USA, pp. 41–45.

Henrysson, A., Billinghurst, M. and Ollila, M. (2005), 'Face to face collaborative AR on mobile phones', In *ISMAR '05: Proceedings of the 4th IEEE/ACM*

International Symposium on Mixed and Augmented Reality, IEEE Computer Society, Washington, DC, USA, pp. 80–89.

Hughes, C.E., Smith, E., Stapleton, C.B. and Hughes, D.E. (2004), 'Augmenting museum experiences with mixed reality', In *KSCE 2004*, November.

Jacobs, M.C., Livingston, M.A. and State, A. (1997), 'Managing latency in complex augmented reality systems', In *I3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, ACM, New York, NY, USA, pp. 49–ff.

Kato, H. and Billinghurst, M. (1999), 'Marker tracking and HMD calibration for a video-based augmented reality conferencing system', In *Augmented Reality, 1999. (IWAR '99) Proceedings. 2nd IEEE and ACM International Workshop on*, pp. 85–94.

Klein, G. and Murray, D. (2007), 'Parallel tracking and mapping for small AR workspaces', In *Mixed and Augmented Reality, 2007. ISMAR 2007. Proceedings. 6th IEEE and ACM International Symposium on*, November, pp. 225–234.

Koller, D., Klinker, G., Rose, E., Breen, D., Whitaker, R. and Tuceryan, M. (1997), 'Real-time vision-based camera tracking for augmented reality applications', In *VRST '97: Proceedings of the ACM symposium on Virtual reality software and technology*, ACM, New York, NY, USA, pp. 87–94.

Kölsch, M. and Butner, S. (2009), 'Hardware considerations for embedded vision systems', In Kisačanin, B., Bhattacharyya, S.S. and Chai, S. (Eds.), *Embedded Computer Vision*, Springer-Verlag, London, UK, Chap. 1, pp. 3–26.

Ledermann, F., Reitmayr, G. and Schmalstieg, D. (2002), 'Dynamically shared optical tracking', In *Augmented Reality Toolkit, The First IEEE International Workshop*.

Lepetit, V. and Fua, P. (2005), 'Monocular model-based 3D tracking of rigid objects', *Found. Trends. Comput. Graph. Vis.*, Vol. 1, No. 1, pp. 1–89.

Lepetit, V. and Fua, P. (2006), 'Keypoint recognition using randomized trees', *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, Vol. 28, No. 9, September, pp. 1465–1479.

Loten, T. and Green, R. (2008), 'Embedded computer vision framework on a multimedia processor', In *Image and Vision Computing New Zealand, 2008. IVCNZ 2008. 23rd International Conference*, November, pp. 1–5.

Lowe, D. (1999), 'Object recognition from local scale-invariant features', In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, pp. 1150–1157.

Luk, W., Lee, T., Rice, J., Shirazi, N. and Cheung, P. (1999), 'Reconfigurable computing for augmented reality', In *Field-Programmable Custom Computing Machines, 1999. FCCM '99. Proceedings. Seventh Annual IEEE Symposium on*, pp. 136–145.

Möhring, M., Lessig, C. and Bimber, O. (2004), 'Video see-through AR on consumer cell-phones', In *ISMAR '04: Proceedings of the 3rd IEEE/ACM International Symposium on Mixed and Augmented Reality*, IEEE Computer Society, Washington, DC, USA, pp. 252–253.

Munshi, A., Ginsburg, D. and Shreiner, D. (2009), *OpenGL ES 2.0 Programming Guide*, Addison-Wesley, p. 2.

Naimark, L. and Foxlin, E. (2002), 'Circular data matrix fiducial system and robust image processing for a wearable vision-inertial self-tracker', In *ISMAR '02: Proceedings of the 1st International Symposium on Mixed and Augmented Reality*, IEEE Computer Society, Washington, DC, USA, p. 27.

Neumann, U. and You, S. (1999), 'Natural feature tracking for augmented reality', *Multimedia, IEEE Transactions on*, Vol. 1, No. 1, March, pp. 53–64.

Ozuysal, M., Fua, P. and Lepetit, V. (2007), 'Fast keypoint recognition in ten lines of code', In *Computer Vision and Pattern Recognition, 2007. CVPR '07. Proceedings. IEEE Conference on*, June, pp. 1–8.

Pouwelse, J., Langendoen, K. and Sips, H. (1999), 'A feasible low-power augmented-reality terminal', In *Augmented Reality, 1999. (IWAR '99) Proceedings. 2nd IEEE and ACM International Workshop on*, pp. 55–63.

Raskar, R., Welch, G. and Chen, W.C. (1999), 'Table-top spatially-augmented realty: bringing physical models to life with projected imagery', In *Augmented Reality, 1999. (IWAR '99) Proceedings. 2nd IEEE and ACM International Workshop on*, pp. 64–71.

Ravikumar, C. (2004), 'Multiprocessor architectures for embedded system-on-chip applications', In *VLSI Design, 2004. Proceedings. 17th International Conference on*, pp. 512–519.

Rekimoto, J. (1998), 'Matrix: a realtime object identification and registration method for augmented reality', In *Computer Human Interaction, 1998. Proceedings. 3rd Asia Pacific*, July, pp. 63–68.

Rolland, J., Davis, L.D. and Baillot, Y. (2001), 'A survey of tracking technology for virtual environments', In Barfield, W. and Caudelli, T. (Eds.), *Fundamentals of Wearable Computers and Augmented Reality*, CRC Press, Mahwah, NJ, Chap. 3, pp. 67–112.

Rowe, A., Goode, A., Goel, D. and Nourbakhsh, I. (2007), *CMUcam3: An Open Programmable Embedded Vision Sensor*, Technical Report RI-TR-07-13, Carnegie Mellon Robotics Institute, May.

Roza, E. (2001), 'Systems-on-chip: what are the limits?', *Electronics Communication Engineering Journal*, Vol. 13, No. 6, December, pp. 249–255.

Sauer, F., Wenzel, F., Vogt, S., Tao, Y., Genc, Y. and Bani-Hashemi, A. (2000), 'Augmented workspace: designing an AR testbed', In *Augmented Reality, 2000. (ISAR 2000). Proceedings. IEEE and ACM International Symposium on*, pp. 47–53.

Schmalstieg, D. and Wagner, D. (2007), 'Experiences with handheld augmented reality', In *Mixed and Augmented Reality, 2007. ISMAR 2007. Proceedings. 6th IEEE and ACM International Symposium on*, November, pp. 3–18.

Skrypnyk, I. and Lowe, D. (2004), 'Scene modelling, recognition and tracking with invariant image features', In *Mixed and Augmented Reality, 2004. ISMAR 2004. Proceedings. Third IEEE and ACM International Symposium on*, November, pp. 110–119.

Smith, R., Piekarski, W. and Wigley, G. (2005), 'Hand tracking for low powered mobile AR user interfaces', In *AUIC '05: Proceedings of the Sixth Australasian conference on User interface*, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, pp. 7–16.

State, A., Livingston, M.A., Garrett, W.F., Hirota, G., Whitton, M.C., Pisano, E.D. and Fuchs, H. (1996), 'Technologies for augmented reality systems: realizing ultrasound-guided needle biopsies', In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, pp. 439–446.

Subbarao, R., Meer, P. and Gene, Y. (2005), 'A balanced approach to 3D tracking from image streams', In *Mixed and Augmented Reality, 2005. Proceedings. Fourth IEEE and ACM International Symposium on*, October, pp. 70–78.

Sutherland, I.E. (1968), 'A head-mounted three dimensional display', In *Proc. Fall Joint Computer Conf.*, Am. Federation of Information Processing Soc. (AFIPS), pp. 757–764.

Ta, D.N., Chen, W.C., Gelfand, N. and Pulli, K. (2009), 'SURFTrac: Efficient tracking and continuous object recognition using local feature descriptors', In *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 2937–2944.

Thomas, B., Piekarski, W., Hepworth, D., Gunther, B. and Demczuk, V. (1998), 'A wearable computer system with augmented reality to support terrestrial navigation', In *ISWC '98: Proceedings of the 2nd IEEE International Symposium on Wearable Computers*, IEEE Computer Society, Washington, DC, USA, p. 168.

Uenohara, M. and Kanade, T. (1995), 'Vision-based object registration for real-time image overlay', In *CVRMed '95: Proceedings of the First International Conference on Computer Vision, Virtual Reality and Robotics in Medicine*, Springer-Verlag, London, UK, pp. 13–22.

Wagner, D. and Schmalstieg, D. (2003), 'First steps towards handheld augmented reality', In *Wearable Computers, 2003. Proceedings. Seventh IEEE International Symposium on*, October, pp. 127–135.

Wagner, D. and Schmalstieg, D. (2007), 'ARToolKitPlus for pose tracking on mobile devices', In *Proceedings of 12th Computer Vision Winter Workshop (CVWW'07)*, February.

Wagner, D. and Schmalstieg, D. (2009a), 'Making augmented reality practical on mobile phones, part 1', *Computer Graphics and Applications, IEEE*, Vol. 29, No. 3, May, pp. 12–15.

Wagner, D. and Schmalstieg, D. (2009b), 'Making augmented reality practical on mobile phones, part 2', *Computer Graphics and Applications, IEEE*, Vol. 29, No. 4, July, pp. 6–9.

Wagner, D., Reitmayr, G., Mulloni, A., Drummond, T. and Schmalstieg, D. (2008a), 'Pose tracking from natural features on mobile phones', In *Mixed and Augmented Reality, 2008. ISMAR 2008. Proceedings. 7th IEEE/ACM International Symposium on*, September, pp. 125–134.

Wagner, D., Langlotz, T. and Schmalstieg, D. (2008b), 'Robust and unobtrusive marker tracking on mobile phones', In *Mixed and Augmented Reality, 2008. ISMAR 2008. Proceedings. 7th IEEE/ACM International Symposium on*, September, pp. 121–124.

Wolf, W., Ozer, B. and Lv, T. (2002), 'Smart cameras as embedded systems', *Computer*, Vol. 35, No. 9, September, pp. 48–53.

Xu, F., Zeng, J.J. and Zhang, Y.L. (2008), 'Design of a DSP-based CMOS imaging system for embedded computer vision', In *Cybernetics and Intelligent Systems, 2008 IEEE Conference on*, September, pp. 430–433.

Zhang, X., Fronz, S. and Navab, N. (2002), 'Visual marker detection and decoding in AR systems: a comparative study', In *Mixed and Augmented Reality, 2002. ISMAR 2002. Proceedings. International Symposium on*, pp. 97–106.

Zhou, F., Duh, H.L. and Billinghurst, M. (2008), 'Trends in augmented reality tracking, interaction and display: A review of ten years of ISMAR', In *Mixed and Augmented Reality, 2008. ISMAR 2008. Proceedings. 7th IEEE/ACM International Symposium on*, September, pp. 193–202.